Lecture Notes
Computer Sc

Elisa Bertino (

ECOOP 20
Object-Ori

Lecture Notes in Computer Science 1850
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Elisa Bertino (Ed.)

ECOOP 2000 – Object-Oriented Programming

14th European Conference
Sophia Antipolis and Cannes, France, June 12-16, 2000
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Elisa Bertino
Universita' di Milano
Dipartimento di Scienze dell'Informazione
Via Comelico 39/49, 20135 Milano, Italy
E-mail: bertino@dsi.unimi.it

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Object oriented programming : 14th European conference ; proceedings /
ECOOP 2000, Sophia Antipolis and Cannes, France, June 12 - 16, 2000.
Elisa Bertino (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ;
Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1850)
ISBN 3-540-67660-0

CR Subject Classification (1991): D.1-3, H.2, F.3, C.2, K.4, J.1

ISSN 0302-9743
ISBN 3-540-67660-0 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag is a company in the BertelsmannSpringer publishing group.
© Springer-Verlag Berlin Heidelberg 2000
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN 10722117 06/3142 5 4 3 2 1 0

Preface

Following a 13-year tradition of excellence, the 14th ECOOP conference repeated the success of its predecessors. This excellence is certainly due to the level of maturity that object-oriented technology has reached, which warrants its use as a key paradigm in any computerized system. The principles of the object-oriented paradigm and the features of systems, languages, tools, and methodologies based on it are a source of research ideas and solutions to many in all areas of computer science. ECOOP 2000 showed a thriving field characterized by success on the practical side and at the same time by continuous scientific growth.

Firmly established as a leading forum in the object-oriented arena, ECOOP 2000 received 109 high quality submissions. After a thorough review process, the program committee selected 20 papers, which well reflect relevant trends in object-oriented research: object modeling, type theory, distribution and cooperation, advanced tools, programming languages. The program committee, consisting of 31 distinguished researchers in object-orientation, met in Milan, Italy, to select the papers for inclusion in the technical program of the conference. Each paper, reviewed by at least 4 reviewers, was judged according to scientific and presentation quality, originality, and relevance to the conference topics. In addition to technical contributed papers, the program included invited presentations. We were fortunate that four distinguished experts accepted our invitation to share with us their views on various aspects of object technology: Ole Lehrmann Madsen (Aarhus University, Denmark) on unified programming languages, Li Gong (Sun Microsystems, USA) on security in distributed object systems, Munir Cochinwala (Telcordia Technologies, USA) on object technologies for advanced communication systems, and finally Alan Kay (Walt Disney Imagineering Research & Development) who gave the banquet speech. The program was complemented by two panels, focusing respectively on aspect-oriented programming and mobile code, and internet security and e-commerce. Apart from the technical program, ECOOP 2000 also offered tutorials, workshops, exhibitions, and demonstrations. Thus, we trust researchers and practitioners gained many insights into the state of the art of object technology.

It is impossible to organize such a successful program without the help of many individuals. I would like to express my appreciation to the authors of the submitted papers, and to the program committee members and external referees, who provided timely and significant reviews. I owe special thanks to Richard van de Stadt for his excellent job in handling the electronic submission and review process of the papers. I also thank Marco Mesiti for his assistance during the program committee meeting. Last, but not least, I would like to thank the general organizing chairs, Denis Caromel and Jean-Paul Rigault, and all organizing committee members for their tremendous work in making ECOOP 2000 a successful conference.

April 2000 Elisa Bertino

Organization

ECOOP 2000 was organized by the University of Nice Sophia Antipolis (UNSA) and the National Institute for Research in Computer Science and Control (INRIA), under the auspices of AITO (Association Internationale pour les Technologies Objets). Within the UNSA, several departments cooperated in the organization: the Computer Science Department, the Information Technology Engineering School (ESSI), the Technology Institute (IUT), and the I3S Laboratory, the latter being associated with both the UNSA and the National Council for Scientific Research (CNRS).

Executive Committee

Organizing Chairs:	Denis Caromel (UNSA)
	Jean-Paul Rigault (ESSI, UNSA)
Program Chair:	Elisa Bertino (University of Milan)
Tutorial Chairs:	Isabelle Attali (INRIA)
	Giuseppe Castagna (École Normale Supérieure)
Workshop Chairs:	Jacques Malenfant (Université de Bretagne-Sud)
	Sabine Moisan (INRIA)
Panel Chairs:	Françoise Baude (UNSA)
	Linda M. Northrop (SEI, CMU)
Demonstration Chairs:	Jacques Farré (ESSI, UNSA)
	Manuel Serrano (UNSA)
Poster Chairs:	Anne-Marie Déry (ESSI, UNSA)
	Mireille Fornarino (ESSI, UNSA)
Exhibit Chairs:	Erick Gallesio (ESSI, UNSA)
	Colette Michel (ESSI, UNSA)
Submission Site:	Richard van de Stadt (University of Twente)
Student Volunteer Chairs:	Bruno Martin (ESSI, UNSA)
	Pascal Rapicault (ESSI, UNSA)
Social Events:	Corinne Jullien (I3S, CNRS)
	Bernard Serpette (INRIA)
Advertising, Website:	Laurent Berger (ESSI, UNSA)
	Jean-Yves Tigli (ESSI, UNSA)
Industrial Relations:	Anne-Marie Hugues (ESSI, UNSA)
Administrative Support:	Zohra Kalafi (UNSA)
	Patricia Lachaume (I3S, CNRS)
	Annick Perles and the ESSI administrative staff
	Dany Sergeant (INRIA)
	Jacqueline Tchobanian (INRIA)

Program Committee

Mehmet Akşit	University of Twente, The Netherlands
Suad Alagić	Wichita State University, USA
Jean-Pierre Briot	Paris 6 University, France
Vinny Cahill	Trinity College of Dublin, Ireland
Luca Cardelli	Microsoft Research, Cambridge, UK
Pierre Cointe	École des Mines de Nantes, France
Serge Demeyer	University of Antwerp, Belgium
Asuman Dogac	Middle East Technical University, Turkey
Theo D'Hondt	Free University, Brussels, Belgium
Carlo Ghezzi	Politecnico di Milano, Italy
Yossi Gil	Technion, Haifa, Israel
Rachid Guerraoui	EPFL, Lausanne, Switzerland
Giovanna Guerrini	University of Genova, Italy
Mehdi Jazayeri	Technical University of Vienna, Austria
Eric Jul	University of Copenhagen, Denmark
Gerti Kappel	University of Linz, Austria
Jørgen Lindskov Knudsen	University of Aarhus, Denmark
Doug Lea	State University of New York, USA
Barbara Staudt Lerner	Williams College, USA
Karl Lieberherr	Northeastern University, USA
Boris Magnusson	Lund Institute of Technology, Sweden
Satoshi Matsuoka	Tokyo Institute of Technology, Japan
Ana Moreira	Universidade Nova de Lisboa, Portugal
Ron Morrison	University of St. Andrews, UK
Rui Oliveira	Universidade de Minho, Portugal
Walter Olthoff	DFKI, Germany
Tamer Oszu	Alberta University, Canada
Jens Palsberg	Purdue University, USA
Markku Sakkinen	Tampere University of Technology, Finland
Francesco Tisato	University of Milano-Bicocca, Italy
Jan Vitek	Purdue University, USA

Sponsoring Organizations

European Research Consortium for Informatics and Mathematics ERCIM

telecom valley

AMADEUS

Sun microsystems

Microsoft Research

GEMPLUS

Referees

Franz Acherman
Ole Agesen
Xavier Alvarez
Davide Ancona
Joaquim Aparício
João Araújo
Ulf Asklund
Dharini Balasubramaniam
Carlos Baquero
Luís Barbosa
Lodewijk Bergmans
Joshua Bloch
Noury Bouraqadi
Johan Brichau
Fernando Brito e Abreu
Pim van den Broek
Kim Bruce
Luis Caires
Giuseppe Castagna
Barbara Catania
Walter Cazzola
Shigeru Chiba
Tal Cohen
Aino Cornils
Erik Corry
Juan-Carlos Cruz
Gianpaolo Cugola
Padraig Cunningham
Christian D. Jensen
Silvano Dal-Zilio
Wolfgang De Meuter
Kris De Volder
Giorgio Delzanno
David Detlefs
Anne Doucet
Rémi Douence
Jim Dowling
Karel Driesen
Sophia Drossopoulou
Stéphane Ducasse
Natalie Eckel
Marc Evers
Johan Fabry
Leonidas Fegaras
Luca Ferrarini
Rony Flatscher
Jacques Garrigue
Marie-Pierre Gervais
Miguel Goulão
Thomas Gschwind
Pedro Guerreiro
I. Hakki Toroslu
Görel Hedin
Christian Heide Damm
Roger Henriksson
Martin Hitz
David Holmes
James Hoover
Antony Hosking
Cengiz Icdem
Yuuji Ichisugi
Anders Ive
Hannu-Matti Järvinen
Andrew Kennedy
Graham Kirby
Svetlana Kouznetsova
Kresten Krab Thorup
Reino Kurki-Suonio
Thomas Ledoux
Yuri Leontiev
David Lorenz
Steve MacDonald
Ole Lehrmann Madsen
Eva Magnusson
Margarida Mamede
Klaus Marius Hansen
Kim Mens
Tom Mens
Isabella Merlo
Marco Mesiti
Thomas Meurisse
Mattia Monga
Sandro Morasca
M. Murat Ezbiderli
Oner N. Hamali
Hidemoto Nakada
Jacques Noye
Deniz Oguz

José Orlando Pereira
Alessandro Orso
Johan Ovlinger
Marc Pantel
Jean-François Perrot
Patrik Persson
Frédéric Peschanski
Gian Pietro Picco
Birgit Pröll
Christian Queinnec
Osmar R. Zaiane
Barry Redmond
Sigi Reich
Arend Rensink
Werner Retschitzegger
Nicolas Revault
Matthias Rieger
Mario Südholt
Paulo Sérgio Almeida
Ichiro Satoh
Tilman Schaefer
Jean-Guy Schneider
Pierre Sens
Veikko Seppänen
Magnus Steinby
Don Syme
Tarja Systä
Duane Szafron
Yusuf Tambag
Kenjiro Taura
Michael Thomsen
Sander Tichelaar
Mads Torgersen
Tom Tourwé
Arif Tumer
Ozgur Ulusoy
Werner Van Belle
Vasco Vasconcelos
Karsten Verelst
Cristina Videira Lopes
Juha Vihavainen
John Whaley
Mario Wolzko
Mikal Ziane
Gabi Zodik
Elena Zucca

Contents

Invited Talk 1

UML

Type Theory

Object Relations

Invited Talk 2

Cooperation and Distribution

Java Run-Time

Invited Talk 3

Optimization

Tools

Towards a Unified Programming Language

Ole Lehrmann Madsen

Computer Science Department, Aarhus University,
Åbogade 34, DK-8200 Århus N, Denmark
Ole.L.Madsen@daimi.au.dk

Abstract. The goal of research in programming languages should be to develop languages that integrates the best of concepts and constructs from the various programming paradigms. We do not argue for a multi-paradigm language, where the programmer alternates between the different paradigms/styles. Instead, we find that the languages of the future should integrate the best available concepts and constructs in such a way that the programmer does not think of multiple paradigms when using a given language. In this paper, we describe to what extent the BETA language has been successful in obtaining a unified style and where more research is needed. In addition to traditional paradigms such as object-oriented-, imperative- functional- and logic programming, we also discuss concurrent programming and prototype-based programming. We discuss language features such as the BETA pattern construct, virtual procedures and classes, higher order classes, methods and functions, part objects, block-structure, and class-less objects.

1 Introduction

Object-orientation has been one of the most successful technologies within the area of software development. From the number of new books and products advertising with OO, one may easily get the impression that object-technology has completely replaced traditional imperative programming in languages like Pascal, and C. This is probably not the case – the C-style of programming is still used a lot and preferred for many types of applications. There are undoubtedly usage's of C that should be replaced by OO, but we think that an imperative language have some qualities that makes it well suited for certain kinds of tasks. Other styles of programming, such as functional- and logic programming have less success in industry and are mainly popular in research communities, but we also believe that these styles have something useful to offer.

The term *programming paradigm* is used to distinguish between different styles or perspectives on programming and imperative-, functional-, logic- and object-oriented programming are often mentioned as example of programming paradigms. According to Budd [6], the term programming paradigm was introduced by Robert Floyd [11]. In [36] there is a further discussion of the term programming paradigm with respect to Kuhn's definition of the term paradigm. Nygaard and Sørgaard claim that the term paradigm is meant to characterize incompatible approaches to science, and that re-

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp.1–26, 2000.
© Springer-Verlag Berlin Heidelberg 2000

searchers within different paradigms are unable to communicate. They point out that there is no such incompatibility between programmers adhering to the different programming paradigm. Therefore, they propose to use the term *perspective* instead of paradigm. Although we may agree with Nygaard and Sørgaard, we will make use of the term programming paradigm since this is a commonly accepted term. We may, however, also use the terms perspective or style.

As mentioned, the term paradigm indicates that there is some form of incompatibility or at least a major difference between the various programming styles. There is indeed a big difference between a pure OO program and a corresponding pure functional program. However, this does not mean that OO and functional programming are incompatible.

We think that most programming paradigms have something useful to offer and the goal of language research should be to develop languages that integrate the best available concepts and constructs from the various paradigms. There has recently been an increasing interest in so-called multi-paradigm languages [6,7] and a number of proposals for multi-paradigm languages have been made. We do not argue for a multi-paradigm language where the programmer may alternate between the different paradigms/styles. Instead, we think that programming languages of the future should integrate the best elements of the paradigms in such a way that the programmer does not think of different paradigms when using a given language.

The language and associated concepts of a person restricts his/her capabilities to understand the world. This is also the case with programming languages – and here we do not distinguish between graphical languages intended for analysis and design such as UML [39] or textual based programming languages such as C++ [41] and Java [1]. When developing software you develop models of the application domain and describe them in some more or less formal language like UML and Java. If you only know Pascal, you model the world in Pascal, and the same is true for UML and Java. Any formal language like UML and Java has limitations to its expressiveness, which means that there are parts of the application domain that cannot be adequately modeled. This is why training in modern programming languages is important – the more knowledge of different language concepts and construct a person knows, the richer is his/her vocabulary for modeling aspects of the application domain.

In the design of BETA [22], a major part of the efforts was to develop a conceptual framework that was richer than the actual language in order to be explicit about the limitations of the language.

We also think that this emphasizes the importance of developing a unified paradigm where the programmer does not think in terms of multiple paradigms. The difficulty in developing such a paradigm and associated language is to avoid a language with a large number of constructs.

The difference between multi-paradigm and unified paradigm may perhaps just be a difference in wording. There are indeed a lot of interesting contributions under the heading of multi-paradigm. We do think, however, that there is a major conceptual difference in multi-paradigm compared to unified paradigm.

In addition to traditional paradigms such as OO-, imperative- functional- and logic programming, we will also discuss process-oriented (concurrent) programming and prototype-based programming based on our experience with using the Self-language.

The BETA language was originally designed to integrate useful concepts and constructs from different programming styles. In order to avoid an explosion in the number of features, it was necessary to unify a number of existing features. The most successful example of this is the unification of abstraction mechanisms such as class, procedure, function, type, etc. into one abstraction mechanism called *pattern*. BETA provides support for imperative programming, concurrent programming and to some extent functional programming, but no support for logic programming. BETA does not support prototype-based programming, although it is possible to describe objects that are not instances of classes.

In this paper, we will discuss to what extent BETA has been successful in integration of features/paradigms and where more research is needed. We discuss language features such as the BETA pattern concept (which unifies classes, procedures, etc.), virtual procedures and classes, higher order classes and methods, part objects, block-structure, class-less objects (anonymous classes), etc.

Although this paper is about programming languages, we think that the issues have implications for modeling and design. One of the advantages of object-orientation is that it provides an integrating perspective on analysis, design, implementation, etc. A central element is OO languages and the associated conceptual framework – a good OO language should be well suited for modeling and design as well as implementation.

Section 2 of this paper contains a further discussion of programming paradigms. Section 3 is a description of elements of the BETA language from the perspective of unification. The primary purpose of this paper is to discuss unification of language features/paradigms, and is not intended to be an introduction to BETA. In section 4, we discuss how to approach a unified programming language.

2 Programming Paradigms

In this section, we will give a brief characteristic of some of the main programming paradigms being discussed in the literature starting with traditional paradigms.

2.1 Traditional Paradigms

Imperative Programming

Imperative programming is the most traditional paradigm and we define it as follows: *In imperative programming, a program is regarded as (partially ordered) sequence of actions (procedure calls) manipulating a set of variables.*

Computers were originally viewed as programmable calculators. A calculator may consist of a number of registers for storing values, and a number of operations such as add, sub, mult, and div for manipulating the registers. A programmable calculator may in addition have a store where new routines may be stored and operations for controlling the flow of execution. A computer may then be thought of as a programmable calculator with a large number of registers, and a large program store. Imperative or procedural programming is thus based on a model of a programmable calculator where data (registers) are manipulated through a sequence of actions.

Imperative programming works well for small programs, but for large programs, it may be difficult to understand the structure of the program, especially to keep track of possible side effects on global data when calling procedures. In order to handle these problems, a number of new language mechanisms have been invented. These include abstract data types[1], modules, and principles such as encapsulation and information hiding.

The development of functional, logic and OO programming may be seen as a response to the problems with imperative programming.

Functional Programming

A large research effort has been directed towards giving a mathematical definition of programming languages. At least in early part of this work, it turned out to be difficult to give an intuitive readable[2] semantics of assignable variables (state) and control flow. In a mathematical definition of a programming language, the semantics of a program is defined in terms of a mathematical function mapping input data to output data. This function is then composed of a number of other functions corresponding to elements of the program. As mentioned above, one problem with imperative programming is that it may be difficult for the programmer to keep track of a large number of variables. It thus seemed a natural consequence to develop programming languages based on mathematical concepts and without the notion of assignable variables.

We have the following understanding of functional programming: *in functional programming, a program is regarded as a mathematical function, describing a relation between input and output*.

In a (pure) functional programming language, there is no notion of assignable variable and the concept of state has been eliminated. The basic elements of a functional programming language are functions, values and types. This may include higher-order functions and types. By higher-order we mean functions and types parameterized by functions and types. Functions and types are usually first-class-values that may be passed around as arguments to functions and returned as values of functions. Standard ML [51] is a good example of a functional programming language.

[1] Abstract data types were originally based on the Simula class concept.

[2] Whether or not a mathematical semantic definition is intuitive and readable is of course not an objective matter. It is obviously easier for a person trained in mathematics to understand such a definition than for the average programmer.

Logic Programming

Logic programming is similar to functional programming in the sense that it is based on mathematical concepts, but instead of functions, equations are used to describe relations between input and output: *in logic programming a program is regarded as a set of equations describing relations between input and output.*

In this sense, it is more general than functional programming. Prolog is the classic example of a logic programming language.

The concept of constraint programming as pioneered by Borning and others [3,12] may be placed within the same paradigm, since it is based on using equations to describe the relationship between data in a program. Rule-based programming is another term often used.

One problem with this paradigm is that it is not possible to provide a solver for equations in general. This means that there have to be restrictions on the kind of equations that can be used in a logic programming language. These restrictions may be difficult to understand for an average programmer.

Object-Oriented Programming

We consider the concept of object-oriented programming to be well understood although it may differ how people might define it. In [23,22], the BETA view on object-oriented programming is defined as follows: *in object-oriented programming a program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world.*

The central part of this definition is the word *physical*. Objects are viewed as *computerized material* that may be used to construct physical models. Objects are viewed as physical material similar to cardboard, sheets and Lego bricks that are materials commonly used to construct physical models by architects, engineers and children. For a further discussion of this definition of OO, see [23,22].

Object-oriented programming may be seen as a development in an opposite direction of functional and logic programming. In the latter, variables and state have been eliminated from the languages. In OO, state has become central in the form of objects.

The success of OO may be because most interesting systems have state and although state can be emulated in functional- and logic programming, it does not appear natural for the average programmer. The conceptual framework underlying object-orientation in terms of classification and composition does also appear more natural than the concepts of mathematics.

2.2 Other Paradigms

Imperative-, functional-, logic- and object-oriented programmings are the most common paradigms being discussed in the literature, but there are other paradigms being discussed:

- **Process-based programming**. In [26] programming in terms of concurrent processes is described as an example of a paradigm. In the extreme where a program is composed of a number of small concurrent processes, of course this style differs from the above-mentioned paradigms.

2. **Block-structured programming**. In [50], block-structure is described as an example of an imperative paradigm.
3. **Prototype-based programming** [20,48] is a paradigm that differs in many ways from the above-mentioned paradigms. This is a programming style that has something useful to offer and should be integrated in future programming languages.

As mentioned we think there are useful elements in most programming styles. In a later section, we will discuss this further, but before doing this we will discuss to what extent, the BETA language has been successful in unifying concepts from different programming styles/languages.

3. BETA

The BETA language was originally designed to integrate useful concepts and constructs from different programming styles – although Simula [8] was the basis for the design of BETA it was certainly a goal to integrate as many concepts and constructs from other languages/paradigms. Of course, this needed to be done in a manner that does not lead to a proliferation of concepts. In [49] Peter Wegner has an interesting critique of Ada with respect to *proliferation of concepts*. He points out a number of differences on how abstraction mechanisms are treated in Ada.

When designing a new language to offer the best of constructs from a number of languages/paradigms, it is necessary to unify and generalize existing constructs. Generalization of a number of similar concepts is a well-known abstraction principle and in the following, we describe how generalization was applied during development of BETA.

This paper is not attempted to be a description of BETA – many details are left out in order to emphasize the important elements of BETA. As most other languages, BETA has some idiosyncrasies. BETA is an old language, so there is room for improvement.

Pattern

An important observation was that most programming languages provide a number of abstraction mechanisms that all have certain common properties. Examples of such abstractions are: procedures as in Algol, classes as in Simula, types as in Pascal, functions as in Lisp, and task types as in Ada. A common characteristic of these abstraction mechanisms is that they describe entities with a common structure and that they may be used as generators for instances of these entities:

- The instances of a procedure are procedure-activation-records.
- The instances of a class are objects.
- The instances of a function are function-activations.
- The instances of types are values.
- The instances of task types are tasks.

In addition to the above mentioned abstraction mechanisms, there are several other examples, such as generic classes, and exception types.

The first goal of BETA was to develop a language construct that could be viewed as a generalization of all known abstraction mechanisms. Concrete abstraction mechanisms could then be viewed as specializations as shown in figure 3.1. Here class, procedure, type, etc. are shown as specializations of the more general abstraction mechanisms: the pattern. The pattern construct is the only abstraction mechanism in BETA. A pattern may be used as a class, procedure, type, etc. A pattern used as a class is often referred to as a class pattern; a pattern used as a procedure (method) is often referred to as a procedure pattern, etc.

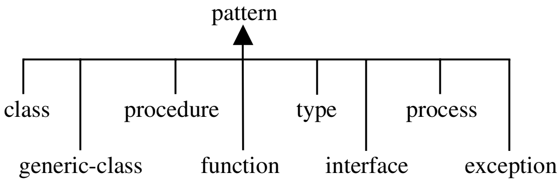


Fig. 3.1.

It is outside the scope of this paper to give a detailed account of BETA. Please consider [22,26,24,28]. We will sketch, however, some incomplete examples to illustrate the main points. In figure 3.2 is shown an example of a 'class' `Point` with three 'operations': `display`, `move` and `add`. The 'class' `Point` and the 'operations' `display`, `move` and `add` are all examples of patterns.

For the sake of simplicity, the data-representation of a `Point` is not shown. In addition, `move` and `add` will have parameters in a complete example. The details of the example are as follows:

```
Point:
  (# display: (# … do … #);
     move: (# … do … #);
     add: (# … do … #);
  #);

S: ^Point;
&Point[]→ S[];
S.display;
```

Fig. 3.2.

- A pattern may contain declarations of attributes in the form of patterns or data-items. `Point` has three pattern-attributes `display`, `move` and `add`. The dots between **(#** and **do** indicate possible attributes of `display`, `move` and `add`.
- A pattern may have a **do**-part, which is a sequence of imperatives (statements) to be executed. The procedure patterns `display`, `move` and `add` have a **do**-part illustrated by the dots following **do**. `Point` has no **do**-part in this example.
- The construct `S: ^Point` is a declaration of a data-item in the form of a reference variable that may refer instances of `Point` or its subpatterns.
- The construct `&Point[]` generates a new (&) instance of `Point` and a reference (`[]`) to this instance is assigned (→) to `S`.
- The construct `S.display` invokes the operation (method) `display` on the `Point`-instance referred by `S`.

The constructs `&Point[]` and `S.display` are two examples of creating instances of a pattern. The value of `&Point[]` is a reference to the new `Point`-instance and corresponds to the new-operator in most object-oriented languages. In `S.display`, a new `display`-instance is generated and executed (by executing its **do**-part). This corresponds to procedure-activation.

Syntactical there is no difference between the declarations of `Point` and say, `display`. The difference is in how these patterns are used. In the example, `Point` is used as a class since we create instances of it. `Display` is used as a procedure, since we create instances that are immediately executed. It is possible to execute `Point` instances and to create references to `display`-objects. In figure 3.3, the variable `d` may refer instances of `S.display` and such an instance is generated and assigned to `d`. This `S.display` instance is then executed by the imperative `d`. Notice the difference between `S.display` and `d`; the former creates and executes a new instance of `S.display`; the latter executes an existing instance. Two subsequent executions of `d` will thus execute the same instance.

```
d: ^S.display
&S.display[]→ d[];
d;
```

Fig. 3.3.

It is possible to invoke `Point` as a procedure pattern but since `Point` has no **do**-part, there will be no actions executed. In the section 3.3 below, an example of a class pattern with a **do**-part is shown.

3.1 Benefits of the Unification

The unification of abstraction mechanisms has the following benefits:

- All abstraction mechanisms are treated in a systematic way, and there is focus on the abstraction-instance relationship.
- Syntactically and conceptually, the language has become simpler.
- There are a number of new technical possibilities. A good unification of concepts/constructs should result in a mechanism that covers the original mechanisms and gives some new possibilities as well. Below we will discuss this.

There may of course also be disadvantages. Abstractions such as class, procedure and exception are still useful concepts. The unification implies that there is no syntactical difference between a class patterns, and procedure patterns, etc. In the original design for BETA, the main goal of the ‘pattern-exercise’ was to obtain a systematical structure for all abstraction mechanisms. The final language might still have special constructs for common abstractions like class and procedure. However, the experience from practice seems to be that there is no demand for such special constructs.

In the following, we will elaborate on the new technical possibilities of the unification.

Subpattern

The subpattern mechanism covers subclasses as in most other languages. That is it is possible to organize class patterns in a usual classification/inheritance hierarchy. In addition, procedure patterns may be organized in a subprocedure hierarchy in the same way as classes may be organized in a subclass hierarchy. This was the subject of an early paper on BETA called *Classification of actions or inheritance also for methods* [18].

Inheritance for procedures is based on the **inner**-mechanism known from Simula. By means of **inner** it is possible to combine the **do**-parts of patterns. In figure 3.4 is shown how a general monitor pattern may be defined by means of **inner**. The pattern monitor has two attributes: entry and mutex. Entry is an abstract procedure pattern that is used as a super procedure in subclasses of monitor. In the buffer class, the operations put and get are subpatterns of entry. Whenever a put- or get-operation is executed, the **do**-part of entry is executed. Execution of **inner** in this **do**-part will then execute the **do**-part of put or get respectively. The **do**-part of entry uses the mutex-attribute to ensure that at most one put- or get-operation is executed at the same time.

```
monitor:
  (# entry:
       (#
       do mutex.P;
          inner;
          mutex.V
       #);
     mutex: @semaphore
  #);
buffer: monitor
  (# put: entry(# do … #);
     get: entry(# do … #)
  #)
```

Fig. 3.4.

In general, procedure inheritance is a powerful mechanism that gives additional possibilities for defining general patterns. The monitor pattern is an example of a concurrency abstraction. Procedure inheritance may also be used to define control abstractions such as iterators.

Standard inheritance for classes may be used to model/represent classification hierarchies, which are well understood. From a modeling point-of-view, we need a similar understanding of classification of actions. For a discussion of this see [18,22,29].

Since patterns may also be used to describe abstraction mechanisms such as functions, coroutines, concurrent processes, and exceptions, these may also be organized in a pattern hierarchy. For examples of this, see [22].

Virtual Pattern

A pattern may be declared virtual in much the same way as a procedure may be declared virtual in Simula [8,32], Eiffel [34], C++ [41] and Java. A virtual procedure pattern is thus similar to a virtual procedure. As mentioned, there is no syntactic difference between class patterns and procedure patterns. A pattern may be used as a class and a virtual pattern may also be used as a class. Such a virtual class pattern provides an alternative to generic classes as in Eiffel or templates as in C++. The concept of virtual class may be applied to other OO languages than BETA [42,16], and may be seen as an object-oriented version of genericity.

In figure 3.5 is shown an example of virtual patterns. The class pattern Set has two virtual pattern attributes. Display is an example of a virtual procedure pattern. Element is an example of a virtual class pattern. The construct :< signals a declaration of a virtual pattern, and the construct ::< signals a further binding of the virtual pattern. In BETA, a further binding of virtual does not imply that the virtual is redefined as in C++, Smalltalk [14], etc. Instead, the virtual is further specialized. Display in

```
Set:
   (# element:< object;
      display:< (# ... do ... #);
      current: ^element
   #);
StudentSet:  Set
   (# element ::< Student;
      display::< (# do current.print #)
   #);
```

Fig. 3.5.

StudentSet is a subpattern of display in Set and the **do**-parts of the two display patterns are combined by means of **inner** as described above.

In pattern Set the variable current is of type element. Since element is declared to be a virtual object, the type of current in Set is object. In StudentSet, the type of current is Student, since element has been further bound to Student. Assuming that Student has a print-attribute, the expression current.print is legal within display of StudentSet.

The concept of virtual classes has been discussed in several papers, and we refer to [22,24,26,31] for further details of virtual patterns.

The unification of abstraction mechanisms into patterns has resulted in a simple extension of the concept of virtual procedures to virtual classes. Further unification/refinement [45,46,10] of the virtual pattern mechanism is however possible.

In addition to procedures and classes, the virtual pattern construct also covers virtually for abstractions such as coroutines, concurrent processes, and exceptions.

Nested Pattern

Algol 60 introduced the notion of block structure, which in Algol 60 means that procedures may arbitrarily nested. Simula extended this to allow nesting of classes and procedures. Block structure has been adapted by many languages since Algol, mainly imperative languages such as Pascal, but is also available in the Lisp-dialect Scheme. It was included in Java 1.1.

In BETA patterns may be nested, which means that procedure patterns as well as class patterns, etc. may be nested. A class with operations, such as the one in figure 3.2 may be considered an example of pattern nesting. In figure 3.2, the patterns `display`, `move` and `add` are nested within the pattern `Point`. In an OO setting, nesting of classes is an interesting aspect and several examples of this have been documented in the literature [25,26,28,29,53].

The class construct makes it possible to *encapsulate* data-items and operations that together constitute a logically related whole. Block structure makes it possible to include classes among items being encapsulated. This makes it possible to place a class in a proper context, just as data-items and procedures may be placed in the context of an object. From a modeling, point-of-view OO languages without block structure can model properties of phenomena in terms of objects with data-items and procedures. With nesting, it is possible to include classes in the properties available for modeling a given phenomenon. In addition to the references mentioned above, see [9] for a good discussion of block structure from a modeling point-of-view.

Nested class patterns make it possible to define a class pattern as an attribute of an object. Often other classes define the structure of an object. Nested patterns make it possible to define such classes where they logically belong. In figure 3.6 is shown an example a class pattern `Grammar` with a nested class pattern `Symbol`. Two `Grammar`-instances, `Self`, and `Java` and declared. The variables, `S1`, and `S2` may refer instances of `Java.Symbol` and `R1` and `R2` may refer instances of `Self.Symbol`. Class `Symbol` is placed in the context of a `Grammar`-object, which makes it possible to express that Java-symbols are different from `Self`-symbols.

```
Grammar:
  (# Symbol: (# ... #);
     ...
  #);
Java: @Grammar;
Self: @Grammar;
S1,s2: ^Java.Symbol;
R1,R2: ^Self.Symbol;
```

Fig. 3.6.

Next, we will show how nested patterns may be used to define objects with several interfaces. In figure 3.7 are shown two interfaces. In BETA, an interface may be viewed as a pattern that only has virtual pattern attributes – similar to an abstract class in C++.

```
Printable:
  (# print:< (#… #);
     separator:< (# … #);
     medium:< (# … #)
  #);
Sortable:
  (# key:< (# …#);
     lessThan:< (# … #)
  #);
```

Fig. 3.7.

In figure 3.8, a class `Person`, implementing the `Printable` and `Sortable` interfaces is shown. The implementation of an interface is made by means of a nested pattern inheriting from the interface pattern. The `Sortable` interface is implemented as the nested pattern, `asPrintable`, which inherits from `Printable`. In figure 3.9 is shown how to obtain a reference to the `Printable` interface of the object `X`. By executing `X.asPrintable` a reference to an instance of `asPrintable` is returned. `P` will then denote this object, which is nested inside the `X`-object. Operations on `X` will then be executed in the context of `X`. This style of programming is used to support the Microsoft Component Model in BETA [38].

```
Person:
  (# name, address,
     asPrintable: Printable
       (# print::<
            (#do    print,name,address,…
#);
          separator::< (# … #);
          medium::< (# … #)
       #);
     asSortable: Sortable
       (# key::< (# … #);
          lessThan::< (# … #);
       #)
  #)
```

Fig. 3.8.

```
X: @Person; P: ^Printable;
X.asPrintable → P[];
P.print;
```

Fig. 3.9.

In Objective C [25] and Java, a class may implement several interfaces. Since all operations are implemented as a flat list of operations, it is necessary to have rules for handling possible name conflicts between operations in the interfaces. By implementing interfaces by means of nested classes, the interfaces are properly encapsulated and isolated from each other. The disadvantage in BETA is, of course, that each interface implementation has to be named.

In [25,53] it is shown that nested classes in many cases may be an alternative to multiple inheritance, although it may not in general replace multiple inheritance.

Pattern Variable

BETA includes the notion of pattern variable. This implies that patterns are first class values that may be passed around as parameters to other patterns. By using pattern variables instead of virtual patterns, it is possible dynamically to change the behavior of an object after its generation. Pattern variables cover procedure variables (i.e. a variable that may be assigned different procedures). Since patterns may be used as classes, it is also possible to have variables that may be assigned different classes, etc.

3.2 Part Objects and Singular Objects

Part Objects. A data-item may be a variable reference corresponding to references in Simula, pointers in C++ or Java. In figure 3.10, the variable `S` is a reference that may refer to instances of `Person` or subpatterns of `Person`. In addition, a data-item may be a part object. A part object is created together with the enclosing object and is a fixed part of this object. The name denoting a part object is therefore constant and cannot be assigned a new object. For this reason it is also *type exact*. The variable `R` in figure 3.10 is an example of a part object.

```
Person: (# … #);
S: ^ Person;
R: @ Person;
```

Fig. 3.10.

Singular objects. In BETA, it is possible to describe objects that are not instances of patterns. In figure 3.11 is shown examples of objects, `Joe` and `Lisa` that are instances of pattern `Student`. In addition is shown an object, `headMaster` that is described directly. The object-symbol `@` means that an object is generated as part of the declaration. As can be seen, it is used in the declarations of `Joe`, `Lisa` and `headMaster`. The syntactic unit `Person(#    #)` following `Student:` is called an *object-descriptor* and describes the structure of objects instantiated from `Student`. An object-descriptor may also describe the structure of a singular object such as `headMaster`. Pattern `Student` is a subpattern of `Person`. In a similar way, the singular object `headMaster` is described as inheriting from `Person`. If the symbol `@` is removed from the declaration of `headMaster`, it would instead be a pattern.

```
Student: Person(# … #);
Joe,Lisa: @ Student;
headMaster: @ Person(# … #)
```

Fig. 3.11.

Singular objects are inspired by similar constructs in other languages:

- Algol 60 has the concept of an *inner block*, which may be viewed as a singular procedure call.
- Simula has the notion of *prefixed block*, which may be viewed as a singular object.
- Pascal has the notion of *anonymous type*, which makes it possible to describe the type of a variable as part of the variable declaration.
- Finally, Java 1.1 has also introduced the notion of *anonymous class* corresponding to BETA's singular objects.

The combination of block-structure and singular objects makes it convenient to express inheritance from part objects. In figure 3.12, a pattern Address is defined. An Address may be used for describing persons and companies. A common approach is then to let Person and Company be subclasses of Address. In this way, Person and Company inherits properties from Address.

```
Address:
  (# street: ...;
     town: ...;
     country: ...;
     print:<
      (#
      do inner;
         {print street,town,country}
      #)
  #)
```

Fig. 3.12.

In BETA, it is also possible to inherit properties from a part object. In Figure 3.14, the Address-properties of a Person are defined by means of the part object adr. In addition to being a part object, adr is a singular object since it is not defined as an instance of a pattern. Finally, the block-structure makes it possible to refer to attributes of the enclosing Person-object from within the binding of the virtual procedure pattern print. The print pattern is extended to print the name attribute of the Person.

```
Person:
  (# name: ...;
     adr: @ Address
       (# print::<
            (#do {print name}#)
       #)
  #)
```

Fig. 3.13.

3.3 Concurrency

Patterns may also describe active objects. In BETA, an active object is an object that has its own thread of execution. An active object may execute as a Simula-style coroutine or in concurrency with other active objects. We shall not give technical details here, but just mention that the **do**-part of an object is the head of the thread. The example in figure 3.2 shows a standard example of a class pattern with operations in terms of procedure patterns. The procedure patterns all have a **do**-part, but class Point does not have a **do**-part. In Figure 3.14, the pattern Producer is an example of a pattern with a **do**-part used like a class in the sense that a named instance (P) is generated.

In the declarations of P and C, the | describes that P and C are active objects. When an object is generated from a pattern, it is generated as either a passive object or an active object. It might be possible to simplify this such that any object can be active.

Semaphores are used as the primitives for synchronization. A semaphore is a low-level mechanism and it should only be used to define higher-level synchronization abstractions. The pattern mechanism together with inner is well suited for defining abstractions like monitor (like the one in figure [3.4]) and Ada-like rendezvous.

```
Producer:
  (#
  do cycle(#do …; E → buffer.put; … #)
  #);
Consumer: (# … do … #);
P: @ | Producer;
C: @ | Consumer;
buffer: @ monitor(# put: …; get: …; #)
```

Fig. 3.14.

4 Towards a Unified Paradigm

In the previous section, it has been shown that BETA to some extent unifies constructs from different programming paradigms. BETA is however not perfect since a number of useful programming concepts are not supported. In this section, we will discuss how to approach a programming language with more unification. The reader will observe that we are heavily influenced by our background in object-orientation and BETA. To obtain real progress in this area, another perspective may be needed.

With the current state of art, we think that a unified programming language should be based on OO. There are a number of reasons for this:

- OO is best suited for the describing the overall structure of large programs - a program should be organized by means of classes and objects.
- Most systems have state that varies over time, so direct support for organization of state is necessary. OO seems best suited for this.
- In addition, there are the usual advantages of OO, such as support for modeling, reuse/extensibility, and as an integrating mechanism for many aspects of the software development process.

In the following, we will discuss features from other paradigms that we would like to include in an integrated paradigm.

A main requirement for a programming language is that there is a simple and natural unification of the various programming styles, such that the programmer does not think of alternating between different paradigms.

4.1 Imperative Programming

Certain algorithms are sometimes best expressed by means of an imperative program. Most OO languages are imperative in the sense that each method is usually written in an imperative style. In Simula and BETA, imperative programming is directly supported since it is possible to write procedures that are not part of a class. As already mentioned, Algol 60 is a subset of Simula.

Block structure may be viewed as a feature from imperative languages, and as argued above, we find that block structure is a useful feature. In addition to nesting of procedures, nesting of classes should be supported.

We find that BETA in its current form does support imperative programming in a satisfactory way, including support for block structure. We do not recommend that the imperative style is used as a primary style, but it should be possible to use for parts of a program.

4.2 Functional Integration

There is a smooth transition from imperative programming to functional programming. As mentioned above, we consider functional programming to be programming without variables that may change their value during a computation. In functional programming, there is no notion of state. A feature such as recursion is mainly associated with functional programming, but in recursion is also heavily applied in imperative programming. Most practical languages provide a mixture of support for imperative as well as functional programming. Lisp is an example of this.

A main disadvantage of procedural programming is that the use of global variables being modified by a set of procedures may lead to programs that are complicated to understand. Functional programming may improve on this. The main benefits of functional programming, as we see it, may be summarized as follows:

- In a functional program, there is no state and variables. This means that functions are pure transformations of input values to output values. A function has no side effect on a global state. This makes it easier to prove the correctness of a given piece of code.
- Recursion is a powerful means in programming – although it may not just be associated with functional programming.
- Functional programming languages often provide higher order functions and types, which have proved to be useful in many cases.
- Function and types are often first class values that may be passed as arguments and returned as values of functions.

We would like to include the positive aspects of these features in a unified programming language and avoid what we consider the negative aspects. For the average programmer, a large functional program may be difficult to understand if there is a heavy use of recursive functions and higher-order types and functions. As for procedural programming, we think that the functional style should be used for programming in the small.

We see functional programming as useful for expressing state transitions within an object. In many cases, the intermediate states obtained through evaluation of a method may not be interesting. The states of the object before and after evaluation of the method are of interest.

A modern programming language should support higher order functions, and types. In accordance with the concept of a pattern as the most general abstraction mechanism, we see functions and types as special variants of patterns just as a class is a special form of a pattern. This implies that if a language supports higher-order functions and types, it should also support higher order classes as well. In general, any abstraction mechanism in the language should be 'higher-order'. If 'higher-order' is not supported in a unified form for all abstraction mechanisms, there will easily be asymmetry between the treatment of abstraction mechanisms such as function, type and class.

We think that it is desirable to be able to handle functions and types as first class values that can be passed as arguments and returned as values of functions. Any abstraction, including class and procedure should then be a first-class-value. In OO languages like Smalltalk and Self, a class is an object and may be passed as a value. A method is, however, as we understand it, not a first-class-value.

If abstractions are first-class-values, it is natural in an imperative setting to be able to have variables that can be assigned such values. This appears to be natural and useful in a language unifying the best from imperative and functional programming.

4.2.1 Functional Programming in BETA

BETA does support some of the requirements for functional programming mentioned in the previous section. The pattern mechanism makes it possible to define subpatterns, nested patterns, variable patterns and virtual patterns. Since patterns may be used as functions, and types, the OO concepts of subclassing and virtuals are also available for the functional style. In addition nesting of functions is possible and functions may be first class values through the pattern variable concept.

In the design of BETA a new syntax for function application was developed. The purpose of this syntax was to unify assignment, procedure call and function application. BETA has an evaluation imperative of the following form:

```
E1 -> E2 ->    -> En
```

Such an evaluation is evaluated from left to right. `E1` is evaluation and its value is assigned to `E2`, which is evaluated giving a value assigned to `E3`, etc. Finally, the result of `En-1` is assigned to `En`, which is evaluated. A simple assignment is an example of an evaluation. The value of an expression `E` may be assigned to a variable `V` in the following way:

```
E -> V
```

Multiple assignments where both of the variables `V` and `W` are assigned the value of `E` are also possible:

```
E -> V -> W
```

Function application may be expressed in the following way:

```
(e1,e2,e3) -> foo -> V
```

A nested function application often has the following syntax in a traditional language:

```
x = F(G(a,b),H(c))
```

In BETA this will be written as follows:

```
((a,b) → G,c->H) → F → x
```

The rationale for the new syntax was to improve readability of nested function calls, and provide a simple unified syntax for assignment, procedure call and function invocation.

A function may return a list of values as shown in the following example:

```
(a,b,c) → F → (x,y)
```

If `G` takes two input arguments, the result of `F` may be passed immediately to `G`:

```
(a,b,c) → F → G → x
```

Limitations of BETA

In BETA, control structures are not evaluations, and this is a limitation for functional programming. In figure 4.1 are shown the control structures[3] of BETA:

```
(if C then I1 else I2 if)
(for i: range repeat Imp for)
L:
  (x … leave L; …
       restart L; …
  x)
```

Fig. 4.1.

- The if-imperative is a standard conditional statement.
- The for-imperative executes `Imp`, `range` number of times.
- An imperative may be labeled. The imperative **`(x … x)`** has the label `L`.
- **`(x … x)`** may be one of **`(# … #)`**, **`(if … if)`** or **`(for … for)`**. One may for example write an imperative of the form:

  ```
  L: (if cond then …; restart L else …; leave L; if)
  ```

- The execution of a labeled imperative may be interrupted by a **`leave`**, which transfers execution to the end of the labeled imperative. A **`restart`** will similarly execute the labeled imperative from its beginning. **`Leave`** and **`restart`** are thus a form of structured goto-statements.

It is of course straightforward to convert the **`if`**-imperative into a conditional evaluation as known from many other languages. This would allow an evaluation of the form:

```
E1 → (if C then E2 else E3 if) → E4
```

resulting in either `E1 → E2 → E4` or `E1 → E3 → E4`. A similar generalization of the for-imperative might look as follows:

```
E1 → (for i: range repeat E2 for) → E3
```

resulting in the following evaluation:

```
E1 → E2 → E2 → ... → E2 → E3
```

[3] The `if`-imperative exits in a more general variant corresponding to a case/switch-statement.

where E2 is evaluated range number of times. For **restart** the corresponding evaluation may look as follows:

```
E1 → L: (x E2 → ... → E3 → restart L; ... x) → E4
```

resulting in the evaluation:

```
E1 → E2 → ... → E3 → E2 → ... → E4;
```

Note that the value of E3 being assigned to restart L is assigned to the evaluation E2 following the label L. Finally a similar form may be used for **leave**:

```
E1 →L: (x E2 → ... → E3 → leave L; ... x) → E4
```

resulting in:
E1 → E2 → ... → E3 → E4

It is, however, not obvious that the above proposals are the best way to approach more unification. Perhaps it will be a better idea to base control structures on the simple idea of methods to Boolean values and/or blocks found in Smalltalk and Self.

Binding of Values to Variables

Functional languages often have a construct like

```
let V = exp1 in exp2
```

where the value of exp1 is bound to the name V and V may denote this value in the expression exp2. This form of assign-one variable is useful in functional languages since it allows intermediate values to be denoted by names without introducing assignable variables in general. The same form of variables is useful in OO languages as well. There is often a need to express that a value is bound to a variable when an object is instantiated or when a method is called. During the lifetime of the object or method-activation, the variable should remain constant. Pascal had the concept of call-by-constant parameters, which support this facility.

In BETA, some of this functionality is available by means of virtual patterns. A value in BETA is considered an abstraction, and consequently a value is represented by a pattern.

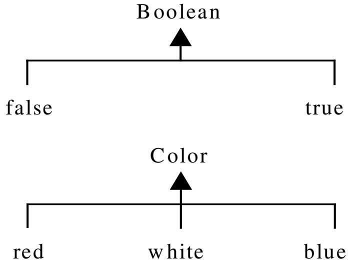


Fig. 4.2.

Since values are considered patterns, values and types may be organized in a subpattern (classification) hierarchy. In general enumerations such as Boolean and color may be organized in a subpattern hierarchy as shown in figure 4.2. In the BETA system, false and true are actually described as subpatterns of Boolean. With respect to enumerations, in general, there is no good syntactic support in BETA. It is possible to emulate enumerations by means of patterns, but a better syntactic support would be desirable. It is easy to introduce a special syntax for this, but no acceptable general syntax has been found.

In line with enumeration values, numbers are also considered patterns, but this is not reflected in the syntax of the language.

In figure 4.3 is shown how to use virtual patterns to represent values that are constantly bound by subpatterns. In instances of `child`, `young` will constantly denote the value true. In order to generalize further, it might be desirable to be able to bind a virtual pattern to an expression (`age < 18`) evaluating to some value as shown in figure 4.4. This is, however, not possible in BETA. In [47] concrete proposals for this are discussed.

```
person: (# young:< boolean; … #);
child: person(# young::< true #);
adult: person(# young::< false #);
```

Fig. 4.3.

```
customer: person
   (# young::< age < 18 #)
```

Fig. 4.4.

The issues of values, objects and patterns are interesting, but non-trivial. In Smalltalk, values are objects and an expression is a combination of method calls to value-objects. In BETA, a value is an abstraction (represented by a pattern) of measurable properties of objects. The discussion of abstraction in [24] has influenced this point-of-view. Here a number, like seven is considered an abstraction over all phenomena that has the property of consisting of seven elements. See [21] for a discussion of the relationships between values and objects. For a further discussion of values in BETA, see chapter 18 in [22].

4.3 Constraint Integration

Logic-and constraint programmings are useful for certain types of problems and we would like to integrate these paradigms as well. We have, however, no concrete proposal for how to do this. In general, we consider constraint programming as an extension of functional programming in the sense that the former is based on functions and the latter on equations. The difficult with equations is, of course, that this requires an equation/constraint solver, which in general is not possible to provide. We have studied with interest earlier proposal such as Primula [40], Loops [2], Leda [6] and the work of Borning & Freeman-Benson [3,12] and the reader is referred to their work for further insight. We would like to be able to use equations whenever convenient. One example is to state the relationship between data-items of an object as shown in Figure 4.5[4]. The constraint `spouse.spouse=self` specifies that the `spouse` of a `Person` has to be the `Person` itself.

```
Person: class   {
   spouse: ref Person;
   constraint spouse.spouse = self
  }
```

Fig. 4.5.

[4] This is not BETA.

4.4 Concurrent Programming

Support for concurrency should be included in a modern programming language. In Simula, any object can act as a coroutine and the basic Simula frameworks includes support for so-called quasi-parallel systems that makes it possible to model real-life systems consisting of a number of interacting processes. Support for concurrency is still an open issue in the sense that there is no common agreement on how to support concurrency in an OO language, despite the fact that a lot of research has been going on in this area. Concurrent Pascal [4] may be considered the first concurrent OO language, since it is based on the class concept from Simula. However, subclassing and virtual procedures were not included in Concurrent Pascal. Concurrent Pascal is one of the first examples of a concurrent programming language and there exist many other proposals. It is outside the scope of this paper to provide adequate references.

The Java language has concurrency and supports protection of shared data by means of monitors. Shared data may be placed in monitors, but there is no language rule that requires shared data to be protected by monitors[5]. This lack of language support for protection of shared data in Java has been heavily criticized by Brinch-Hansen [5].

As mentioned, there does not seem to a general agreement for a concurrency model for OO languages. This is probably because there is no common model that is well suited for all purposes. We think that the following requirements should be made for a concurrent programming language:

1 Concurrency should be an integrated part of the language.

2 The language should provide basic primitives that make it possible to define high-level concurrency abstractions, such as monitors, Ada-language rendezvous, etc.

3 It must be possible to write schedulers – In Simula, and BETA it is possible to do this and this has been used extensively in the Simula community [32,33].

4 There should be language support for ensuring protection of shared variables. As pointed out by Brinch-Hansen it is too easy with languages like Java (and BETA) to write concurrent processes that incidentally access the same shared data. The Concurrent Pascal model is probably too restrictive, but it is not obvious how to relax on this and at the same time avoid the weaknesses of Java and BETA.

5 We also think that further work is needed with respect to better conceptual means for modeling concurrency. In [22], a distinction between alternation and real concurrency is made in order to capture the difference between preemptive and non-preemptive scheduling from a modeling perspective.

6 For distributed computing, we think that there is an even higher demand for development of a new conceptual model. The current conceptual framework for OO is primarily based on the ideas from Simula. For distributed computing, it is necessary to take time and location into account. A possible direction for this may be found in [35].

[5] This is also the case for BETA.

4.5 Prototype-Based Programming

Prototype-based languages are one of one of the most interesting developments within object-orientation. Prototype-based languages make it possible to program without classes, but they do not in general support programming using a class-like style. We think that a unified language should support class-based as well as prototype-based programming.

Class-based programming is based on a modeling perspective where the real world is conceived in terms of phenomena and concepts. Additional conceptual means such as composition and classification are used to organize our understanding of knowledge from the application domain. Phenomena and concepts are then represented by means of objects and classes (patterns in BETA). Classification is supported by the class/subclass construct, which makes it possible to represent hierarchies of concepts. Composition is supported by constructs such as instance variables, part objects and nested patterns (classes, procedures, etc.), but most OO languages do not provide as explicit support for composition as they do for classification. The conceptual framework for BETA is further described in [23,22].

Prototype-based programming is based on a modeling perspective where new phenomena are identified in terms of their similarity with well-known phenomena, so-called prototypes. The standard example [20] is the elephant Clyde that represents a prototypical elephant in the mind of some person. If this person encounters a new situation with another elephant, say one named Fred, facts about Clyde can be used for understanding properties about Fred.

The rationale behind this modeling perspective is that this is how children perceive the world and that people in general perceive new phenomena in this way. Programmers often work this way: new code is often started by copying some existing code, which is then modified to adapt to the new requirements and often the relationships with the original source are lost. With traditional programming languages, there is no support for this copy-paste style of programming. In prototype-based languages, this is the main style and there is direct support for expressing the relationships between the source object and the new object.

We think that prototype-based programming is useful in the exploratory phases of programming. It is very useful to be able to copy objects and modify parts of their functionality without having to define classes and instances. We also think that prototype-based programming is well suited for teaching introductory programming. Objects are concrete entities that are easier for beginners to relate to than abstract entities such as classes.

The strength of prototype-based programming is also its weakness. New phenomena may be conceived according to a prototypical style, but in order to organize knowledge about a complicated application domain, we develop concepts for grasping the interesting properties of phenomena. In addition, classification and composition are applied to structure our knowledge. In programming, the same happens. After an exploratory phase, we need to organize a program into suitable structures.

In a unified programming language we think that concepts and constructs from prototype-based languages should be available for supporting the *exploratory* phases of programming. The mechanism from class-based languages should be available for supporting the *structural* phases where objects are organized into structures. We think that this is desirable from a modeling as well as a technical point-of-view.

The main programming language constructs of prototype-based languages are copying of objects, dynamic modification of objects, and delegation. The literature contains a number of papers discussing to what extent class-based programming can be realized in prototype-based languages. The class-based style can obviously be emulated in a prototype-based language, but we would prefer more direct support for the class-based style than is currently possible. Some interesting work in this direction has recently been published.

For a further discussion of prototype- versus class-based programming, see [29].

4.6 Other Possibilities for Unification

In [29] we discuss other issues that may be candidates for a unified language:

1. **Multi-methods versus single methods**. Multi-methods are a generalization of single dispatch methods, and may be a candidate for a unified language.
2. **Class versus module**. BETA does not unify pattern and module, since modularization of code is considered independent of abstraction in the application domain. This is in contrast to a language like Eiffel, where a class and a module are the same.
3. **Graphical design languages versus text-based programming languages.** Regarding graphical design languages, we think that there are no good reasons to distinguish a design language like UML from a programming language like Java. There should be no major differences in the underlying abstract languages. There are many good reasons to provide a graphical syntax for part of a programming language, but there should be an immediate mapping from the graphical language to the programming language. For the static structure of an object model expressed, by class, subclass, composition (aggregation), virtual functions, etc. there are no problems in obtaining this. For associations as supported by UML and ODMG, there might be better support in programming languages; see [26] for a discussion of this. For dynamic aspects, programming languages do in general not have good support for state machines. In [30] direct support for state machines in BETA has been suggested.

5. Conclusion

The main conclusion of this paper is that programming languages of the future should support a unified perspective on programming including the best concepts and constructs from the current programming languages and paradigms. In this paper we have argue that most programming paradigms have something useful to offer. We stress the importance of unified-paradigm instead of multi-paradigm, since we think that it is important that the programmer does not think of multiple paradigms when using a given language.

We think that BETA is a step in the right direction. The pattern construct was intended to be the ultimate abstraction mechanism, but there is still a way to go before this can be obtained. In addition to the issues discussed above, there are the issues of language mechanisms for capturing abstractions as expressed by design patterns [13] and other types of patterns. The technical meaning of the word pattern is of course different when used as a BETA language construct or when used by the pattern community. However, conceptually in both cases pattern cover abstractions of program elements.

There may be tasks/domains that require their own paradigm/language, although we think that frameworks perhaps with a special syntax may handle this. The interesting and useful challenges for the future are to provide languages that in a unified way offer the best of all styles.

Acknowledgements. BETA was designed by Bent Bruun Kristensen, Birger Møller-Pedersen, Kristen Nygaard and the author. The work presented here has evolved though discussions with numerous colleagues and students in Århus and other places, including Jørgen Lindskov Knudsen, Boris Magnusson, Randy Smith, and Dave Ungar. The inspiration to focus on unified-paradigm instead of multi-paradigm was the result of the author's participation in a panel on multi-paradigm at ECOOP'98 together with Tim Budd, James Coplien (the organizer), and Ulrich Eisenenecker. I am grateful to Elisa Bertino for inviting me to give a talk at ECOOP'2000 and to write this paper. The paper touches on a wide number of issues and we apologize for the lack of a number of references to related work.

References

1. Arnold, K., Gosling, J.: The Java Programming Language. Addison Wesley, 1996.
2. Bobrow, D.G., Stefik, M.: The LOOPS Manual. Xerox Corporation, Palo Alto, CA, 1983.
3. Borning, A.: The Programming Language Aspects of Thinglab, a Constraint-Oriented Simulation Laboratory. ACM Trans. on Programming Languages and Systems, 3(4):353-387, October 1981.
4. Brinch-Hansen, P.: The Programming Language Concurrent Pascal. IEEE Trans Software Engineering, 1(2), (1975), 149-207.
5. Brinch-Hansen: P.: Java's Insecure Parallelism. ACM SIGPLAN, Vol. 34, No. 4, April 1999.
6. Budd, T.A.: Multiparadigm Programming in Leda. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
7. Coplien, J.: Multi-Paradigm Design for C++. Addison Wesley Longman, 1998.
8. Dahl, O.-J., Nygaard, K., Myrhaug, B.: Simula 67 Common Base Language. Technical Report Publ. no. S-2, Norwegian Computing Center, Oslo, 1968.

9. Ernst, E.: **gbeta** – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance. Ph.D. Thesis, Computer Science Department, Aarhus University, 1999.
10. Ernst. E.: Propagating Class and Method Combination. In: Guerraoui, R. (ed.): 13th European Conference on Object-Oriented Programming, Lisbon, June 1999. Lecture Notes in Computer Science, Vol. 1628. Springer-Verlag, Berlin Heidelberg New York, 1999.
11. Floyd, R.W.: The Paradigms of Programming. Comm. of the ACM, 22(8), 445-460, August 1979.
12. Freeman-Benson, B.N., Borning, A.: Integrating Constraints with an Object-Oriented Language. In: Madsen, O.L. (ed.): 6th European Conference on Object-Oriented Programming, Utrecht, June/July. Lecture Notes in Computer Science, Vol. 615. Springer-Verlag, Berlin Heidelberg New York, 1992.
13. Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.: Design Patterns: Elements of Object-Oriented Software Architecture. Addison-Wesley, 1994.
14. Goldberg, A., Robson, D.: Smalltak-80: The Language and its Implementation. Addison-Wesley Publishing Company, 1983.
15. Hoare, C.A.R.: Notes on Data Structuring. In: Dahl, O.-J., Dijkstra, E., Hoare, C.A.R.: Structured Programming. Academic Press, 1972.
16. Igarashi, A., Pierce, B.C.: Foundations for Virtual Types. In: Guerraoui, R. (ed.): 13th European Conference on Object-Oriented Programming, Lisbon, June 1999. Lecture Notes in Computer Science, Vol. 1628. Springer-Verlag, Berlin Heidelberg New York, 1999.
17. Knudsen, J.L., Löfgren, M., Madsen, O.L., Magnusson, B.: Object-Oriented Environments – The Mjølner Approach. Prentice Hall, 1994.
18. Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B. Nygaard, K.: Abstraction Mechanisms in the BETA Programming Language. In Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages. Austin, Texas, January 1983.
19. Kristensen, B.B., Madsen, O.L., Møller-Pedersen, B. Nygaard, K.: Classification of Actions or Inheritance also for Methods. In: 1st European Conference Object-oriented programming, Paris, France, June 1987. Lecture Notes in Computer Science, Vol. 276. Springer-Verlag, Berlin Heidelberg New York, 1987.
20. Liebermann, H.: Using Prototypical Objects to Implement Shared behavior in Object-Oriented Systems. Proc. OOPSLA'86, Portland, OR, 1986.
21. MacLennan, B.J.: Values and Objects in Programming Languages. ACM SIGPLAN Notices 17(1982) 12(Dec), 70-79.
22. Madsen, O.L., Møller-Pedersen, B., Nygaard, K.: Object-Oriented Programming in the BETA Programming Language. Addison Wesley/ACM Press, Wokingham, England, 1993.
23. Madsen, O.L., Møller-Pedersen, B.: What Object-Oriented Programming may be and What it does not have to be. In: Gjessing, S., Nygaard, K. (eds.): 2nd European Conference on Object-Oriented Programming, Oslo, August 1988. Lecture Notes in Computer Science, Vol. 322. Springer-Verlag, Berlin Heidelberg New York, 1988.
24. Madsen, O.L., Møller-Pedersen, B.: Virtual Classes, a Powerful Mechanism in Object-Oriented Languages. In: Proc. OOPSLA'89, New Orleans, 1989.
25. Madsen, O.L., Møller-Pedersen, B.: Part Objects and their Locations. In: (Magnusson, B., Meyer, B., Perrot, J.F. (eds.): Proc. Technology of Object-Oriented Languages and Systems – TOOLS10. Prentice-Hall, 1993, pp. 283-297.
26. Madsen, O.L., Magnusson, B., Møller-Pedersen, B.: Strong Typing of Object-Oriented Languages Revisited. In: Proc. OOPSLA'90, Ottawa, Canada, 1990.

27. Madsen, O.L.: Block Structure and Object-Oriented Languages. In: Shriver, B.D., Wegner, P. (eds.): Research Directions in Object-Oriented Programming. Cambridge MA: MIT Press, 1987.

28. Madsen, O.L.: An Overview of BETA. In [17].

29. Madsen, O.L.: Open Issues in Object-Oriented Programming - A Scandinavian Perspective. Software Practice and Experience, Vol. 25, No. S4, Dec. 1995.

30. Madsen, O.L: Toward Integration of State Machines and OO Languages. In: Mitchell, R. et al.: (eds.): Technology of Object-Oriented Languages and Systems 29, Nancy, June 1999. IEEE Computer Society, 1999.

31. Madsen, O.L.: Semantic Analysis of Virtual Classes and Nested Classes. In: Proc. OOPSLA'99, Denver, Colorado, 1999.

32. Magnusson, B.: An Overview of Simula. In [17].

33. Magnusson, B.: Personal Communication, 1999.

34. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, 1988.

35. Nygaard, K.: GOODS to Appear on the Stage. In: Aksit, M., Matsouka, S. (eds.): 11th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, Vol. 1241. Springer Verlag, Berlin Heidelberg New York, 1997.

36. Nygaard, K. Sørgaard, P.: The Perspective Concept in Informatics. In Bjerknes, G., Ehn, P., Kyng, M. (eds.): Computers and Democracy. Abury, Aldershot, UK, 1987.

37. The Mjølner System, `http://www.mjolner.com.`

38. Rogerson, D.: Inside COM - Microsoft's Component Object Model, Microsoft Press, 1997.

39. Rumbaugh, J. Jacobsen, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison-Wesley, 1999.

40. Salvesen, A., Wang, A.: Primula – An Integration of Prolog and Simula. Report no. 781, Norwegian Computing Center, 1986.

41. Stroustrup, B.: The C++ Programming Language. Addison-Wesley, 1986.

42. Thorup, K.K.: Genericity in Java with Virtual Types. In: Aksit, M., Matsouka, S. (eds.): 11th European Conference on Object-Oriented Programming. Lecture Notes in Computer Science, Vol. 1241. Springer Verlag, Berlin Heidelberg New York, 1997.

43. Thorup, K.K.: Objective-C. In: Zamir, S. (ed.): Handbook of Object Technology. CRC Press, 1999.

44. Thorup, K.K.: Genericity in Java with Virtual Classes. In: Aksit, M., Matsuoka, S.: (ed.): 11th European Conference on Object-Oriented Programming, Lisbon, June 1997. Lecture Notes in Computer Science, Vol. 1241. Springer-Verlag, Berlin Heidelberg New York, 1997.

45. Thorup, K.K., Torgersen, M.: Unifying Genericity - Combining the Benefits of Virtual Types and parameterized Types. In: Guerraoui, R. (ed.): 13th European Conference on Object-Oriented Programming, Lisbon, June 1999. Lecture Notes in Computer Science, Vol. 1628. Springer-Verlag, Berlin Heidelberg New York, 1999.

46. Torgersen, M.: Virtual Types are Statically Safe. In: Bruce, K. (ed.) 5th Workshop on Foundations of Object-Oriented Languages, (San Diego, CA, January 16-17, 1998).

47. Torgersen, M.: Unifying Abstraction. "Progress Report" in partial fulfillment of the requirements for the Ph.D. degree. Computer Science Department, Aarhus University, 1999.

48. Ungar, D., Smith, R.B.: SELF – The Power of Simplicity. In: Proc. OOPSLA'87, Orlando, FL, 1987.

49. Wegner, P.: On the Unification of Data and Program Abstraction in Ada. In Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages. Austin, Texas, January 1983.

50. Wegner, P.: Concepts and Paradigms of Object-Oriented Programming. Object-Oriented Messenger 1, 1 August 1990.
51. Wikstrøm, A.: Functional Programming using Standard ML. Prentice-Hall NJ, Englewood Cliffs, 1987.
52. Østerbye, K.: Associations as a Language Construct. In: Mitchell, R. et al.: (eds.): Technology of Object-Oriented Languages and Systems 29, Nancy, June 1999. IEEE Computer Society, 1999.
53. Østerbye, K.: Parts, Wholes and Sub-Classes. In Proc. European Simulation Multiconference, ISBN 0-911801-1, 1990.

Creative Object-Oriented Modelling: Support for Intuition, Flexibility, and Collaboration in CASE Tools

Christian Heide Damm, Klaus Marius Hansen, Michael Thomsen, and Michael Tyrsted

Department of Computer Science, University of Aarhus,
Aabogade 34, 8200 Aarhus N, Denmark
{damm,marius,miksen,tyrsted}@daimi.au.dk

Abstract. A major strength in object-oriented development is the direct support for domain modelling offered by the conceptual framework underlying object-orientation. In this framework, domains and systems can be analysed and understood using models at a high level of abstraction. To support the construction of such models, a large number of Computer-Aided Software Engineering tools are available. These tools excel in supporting design and implementation, but have little support for elements such as creativity, flexibility, and collaboration. We believe that this lack of support partly explains the low adoption of CASE tools. Based on this, we have developed a tool, Knight, which supports intuition, flexibility, and collaboration by implementing gesture based UML modelling on a large electronic whiteboard. Such support improves CASE tools, and can thus potentially lead to increased adoption of CASE tools and thus ultimately help improving the overall quality of development projects.

1 Introduction

Object-oriented programming languages provide many technical qualities but more importantly object-oriented languages and object-oriented development in general also provide a conceptual framework for understanding and modelling [19] [21]. This conceptual framework provides abstraction mechanisms for modelling such as concepts (classes), phenomena (objects), and relations between these (inheritance, association, composition) that allow developers to describe *what their system is all about* and to formulate solutions on a higher level of abstraction than program code.

In practice, developers build models at several abstraction levels. The program code can be seen as an executable model, but for purposes such as analysis, specification, documentation, and communication, models visualised graphically in the form of diagrams are often used. The Unified Modeling Language (UML [31]) is a prominent example of a graphical notation that is used for such purposes.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp.27–43, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The so-called Computer-Aided Software Engineering tools (CASE tools) provide tool support for modelling. These can among other things help users to:

- create, edit, and layout diagrams,
- perform syntactic and semantic checks of diagrams and simulate and test models,
- share diagrams between and combine diagrams from several users,
- generate code or code skeletons from diagrams (forward engineering) and generate diagram or diagram sketches from code (reverse and round-trip engineering), and
- produce documentation based on diagrams and models.

But even though CASE tools offer these many attractive features, they are in practise not widely and frequently used [1][15][18]. CASE tools clearly support design and implementation phases, but have less support for the initial phases, when the focus is on understanding the problem domain and on modelling the system supporting the problem domain. To rectify this problem, we propose inclusion of support for intuition, flexibility, and collaboration, and that CASE tools should have more direct, less complex user-interfaces, while they should still preserve the current support for more technical aspects such as implementation, testing, and general software engineering issues.

We have implemented a tool, Knight, which acts as an addition to existing CASE tools. It uses a large electronic whiteboard (Fig. 1) to facilitate co-operation and a mixture of formal and informal elements to enable creative problem solving. This paper discusses the support of the Knight tool for modelling through gesture based creation of UML diagrams.

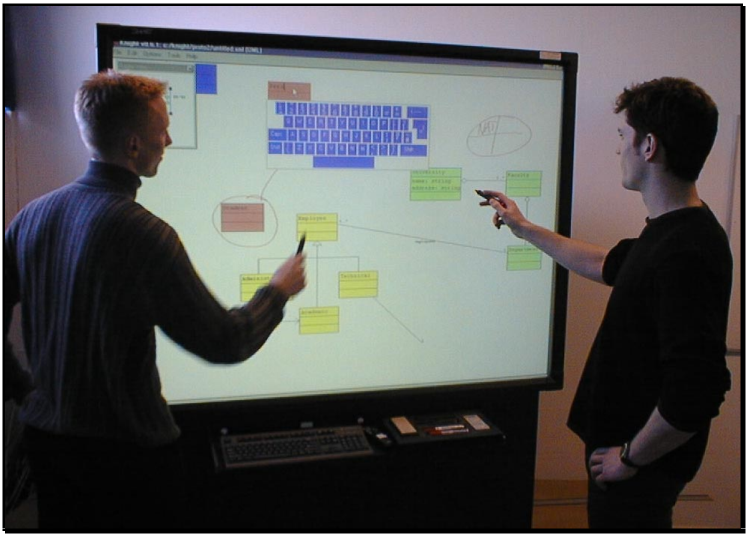


Fig. 1. Use of Knight on an electronic whiteboard

Section 2 gives a theoretical overview of modelling, and section 3 discusses related work on current tool support for object-oriented modelling and the adoption of CASE tools. Section 4 presents studies of modelling in practice and section 5 discusses the Knight tool. Finally, section 6 discusses future work and section 7 concludes.

2 Modelling and Interpretation

Simplistically, object-oriented software development can be viewed as mapping a set of real-world phenomena and concepts to corresponding objects and classes [19][5]. The set of real-world phenomena and concepts is called the *referent system* and the corresponding computerised system is called the *model system* [19] (Fig. 2).

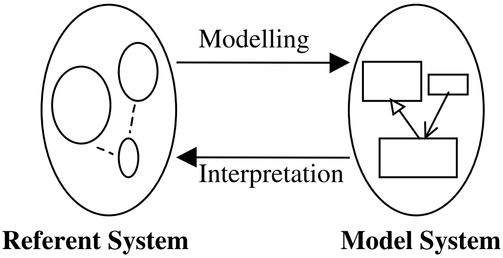


Fig. 2. Modelling and interpretation

Mapping a referent system to a model system is called *modelling*. Modelling is often iterative and it is thus important to be able to discuss a model system in terms of the referent system. We call this reverse process *interpretation* [5]. Modelling is concerned with expressing an understanding of a referent system in a fluent, formal, and complete way, for which usable and formal notations (such as diagramming techniques and programming languages) are crucial. Interpretation is concerned with understanding a model system in terms of the referent system. For doing this, it is important that central concepts in a model system are understandable in the referent system.

Neither understanding the referent system nor building a model system is unproblematic. Understanding the referent system is within the domain of user-oriented disciplines such as ethnography [12] and participatory design [2][11]. The ethnographic perspective on system development is concerned with basing system design on actual work performed within the referent system. The rationale is to avoid a major cause of system failure, namely that the developed system does not fit the work practice within which it is to be used [12]. The participatory design perspective on system design is concerned with designing systems in active collaboration with potential users. This is done for both a moral and a practical reason: Users have to work with the future system, and users are competent practitioners who are knowledgeable of the referent system. Building the model system is within the domain of object-oriented software engineering. Software engineering is concerned with building robust, reliable, and correct systems. In iterative development, then, co-ordination, communication, and collaboration between different perspectives and individual developers is crucial [5]. Thus, when considering tool support for a system development process, tool support for the modelling and the interpretation processes is important.

In iterative development, modelling and interpretation are interleaved. When different competencies work together, it is crucial that both processes are, to a certain extent, understood by all involved parties. For example, when doing user involvement it is both important that the developers understand the work of the users (i.e., the referent system) and that the users are able to react on and help design the application (i.e., the model system.)

3 Related Work on Tool Support

A large variety of tools that support software development, and modelling in particular, exist. In [28], Reiss presents an overview of 12 categories of software tools. CASE tools, or *Design Editors* as they are also referred to, are tools that let a user design a system using a graphical design notation, and that usually generate at least code skeletons or a code framework.

A large body of literature discusses the lack of adoption of CASE tool. In their study of CASE tool adoption [1], Aaen et al. showed that less than one fourth of the analysts in the companies studied used CASE tools and that about half of the respondents had used the tools for two projects or less. In [15], Kemerer reported that one year after the introduction of a CASE tool, 70 % of them were never used again.

In [13], Iivari confirms, and explains, the low adoption and use of CASE tools. Two of the main conclusions of the study are that high CASE tool usage does indeed increase productivity, but that a high degree of voluntariness in using the CASE tool tends to decrease the use of the tool. Important for our concern, the study also disclosed that many developers found it hard to appreciate CASE tools as they often were perceived as having a high complexity. From this, Iivari concludes that any intelligent means of reducing the perceived complexity might be a very profitable investment. Thus, the study to some extent supports our claim that the user-interface of CASE tools needs to be less complex and more direct.

A recent study by Lending and Chervany [18] examines a number of aspects of use of CASE tools. It examines whether CASE tools are perceived as being useful, and if developers using CASE tools use the same methodologies and perform the same activities as developers who do not. Finally, the study points to the features of CASE tools that are being used. Surprisingly, the participants who did use CASE tools, were quite neutral when it came to the perceived usability of CASE tools. Also, the study showed that users of CASE tools on average spent more than twice as much of their time doing analysis, and considerable less time on programming, test, and maintenance, than non-users did. Unfortunately, the study did not show whether this increased time spent on analysis actually resulted in a better analysis model, or whether it was due to overhead caused by the CASE tool.

The respondents indicated that only a small amount of the functionality offered by the CASE tool was actually used. They all used the facilities for creating and editing models and diagrams but used little else. The only other functionality that was used at least “sometimes” was functionality to detect inconsistencies in diagrams, and to create documentation. The study in this way highlights that a focus on increasing the support for creating and editing diagrams in CASE tools is appropriate, and perhaps even the most important support.

In [14], Jarzabek and Huang present the view that current CASE tools are far too focused on hard aspects of software development such as software engineering. In their opinion, softer aspects such as support for creativity and idea generation are needed if CASE tools are to gain a wide acceptance. Concretely, they believe that CASE tools should allow the developer more freedom in expressing ideas and that their environment should be more tolerant, and allow the developers to think and work "at the level of application end users". Also the authors argue that current CASE tools are too method-oriented, and that they should provide a more natural process-oriented framework. We concur with Jarzabek and Huang, and try to support creativity, through enhanced and more flexible support for modelling.

Usage of whiteboards as a support in creative, problem-solving meetings has been studied in several contexts [3][23]. Electronic whiteboards have consequently been used as a computational extension of traditional whiteboards. A goal in systems running on electronic whiteboards has often been to preserve desirable characteristics of whiteboards such as light-weight interaction and informality of drawings [26]. To our knowledge no one has investigated this use in the setting of Computer-Aided Software Engineering. We try to extend the box of tools for electronic whiteboards with a tool supporting object-oriented modelling.

4 Modelling and Interpretation in Practice

We have empirically studied the practice of modelling – and thus interpretation – in three distinct situations with three different user groups. The main results of each of the studies are given below. In each situation, the groups contained a mixture of competencies such as ethnographers, participatory designers, experienced and inexperienced developers, and end users. For a more detailed account, refer to [6].

4.1 The Dragon Project: Designing a New System

This study was carried out informally during the Dragon Project [5]. The project involved participants from a university research group and a major shipping company. Development in the Dragon project took place in active collaboration with users in the sense that at least one user was co-located with the development group at any given time. Whenever modelling of major conceptual areas of the shipping domain took place the users participated actively in this. Actual modelling in these sessions almost always took place on a whiteboard. Although most information from the users was in the form of domain knowledge as verbal or written accounts, drawings were often made by the end user on, or in connection to, the diagrams on the whiteboard. User drawings were informal, in contrast to formal UML models, which described key concepts or relationships from the shipping domain. Eventually, the users nevertheless picked up parts of the UML notation and commented on, e.g., multiplicities on an association.

4.2 COT: Reengineering an Existing Application

We have studied a technology transfer project (COT, `http://www.cit.dk/COT`) involving a university research group and an industrial partner. The project reengineered an existing industrial application for control of a flow meter, while the inexperienced developers from the industrial partner learned object-oriented analysis and design. For the modelling, this project used a mixture of CASE tools, projectors, and whiteboards.

Simplistically, the inexperienced developers explained the existing implementation, whereas the experienced developers modelled a reengineered version. Eventually, this balance shifted as the inexperienced developers picked up larger parts of the UML and participated in the modelling. This introduced a certain number of syntactical errors in the use of the UML. These errors were either repaired, if they disturbed the shared understanding of what was modelled, or ignored, if the meaning was clear from the context. After each modelling session, photographs of the diagrams were taken for future record or for manual entry into a CASE tool. Programming was then done in an ordinary editor and the CASE tool was used to reengineer code into diagrams.

4.3 Mjølner: Restructuring an Existing Application

This study investigated the redesign of the Mjølner integrated development environment (`http://www.mjolner.com`). The group performing the redesign consisted of six developers with different experience in the domain of the integrated development environment. All developers had experience in object-orientation and a fair understanding of UML. Two of the developers had an in-depth knowledge of the development environment, other two developers had a knowledge of the part of the tool that they developed, and the last two developers were introduced to the software architecture of the environment while participating in the redesign.

During the redesign session, the most used artefacts were a whiteboard and a laptop. The whiteboard was used to draw the software architecture of the existing environment and to edit these drawings. The laptop was used whenever a developer needed to look at code in order to remember the actual architecture. This use took place whenever another person was at the whiteboard.

Although almost everything they drew was in actual UML notation, the notation was tweaked in three ways. First, UML did not suffice to explain certain aspects of the architecture leading to informal drawings of this. Second, the language used to implement the environment is BETA [19], which has a number of language and modelling constructs not supported by the UML. Two of these constructs, inner and virtual classes, were used heavily in the implementation, and thus the developers invented new notational elements on the fly. Third, the information drawn was filtered in the sense that often only important attributes, operations, and classes were shown.

4.4 Key Insights

From our analysis of the user studies a number of lessons on the co-ordinative, communicative, and collaborative aspects of object-oriented modelling can be learned. We group the insights into the three categories 'tool usage', 'use of drawings', and 'collaboration'.

Tool Usage. Typically, a diagram existed persistently in a CASE tool as well as transiently on a whiteboard. CASE tools were primarily used for code generation, reverse engineering, and documentation whereas whiteboards were used for collaborative modelling and idea generation. This mix caused a number of problems: Whereas whiteboards are ideal for quickly expressing ideas collaboratively and individually, they are far from ideal for editing diagrams etc. This means that in all user studies, drawings have been transferred from whiteboards to CASE tools and from CASE tools back to whiteboards.

Use of Drawings. Most of the drawing elements were in the form of elements from UML diagrams such as class, sequence, and use case diagrams. However, these elements were combined with non-UML elements in two forms. Either as rich "freehand" elements that explained part of the problem domain or as formal additions to the UML such as notations for inner classes or grouping. Timings from one of the user studies show that approximately 25% of the meeting was spent on actual drawing on the whiteboard. The drawing time was divided into 80% for formal UML diagrams and 20% for incomplete or informal drawings.

Another key observation is the use of filtering. Filtering was used for several reasons. First, even whiteboard real estate is limited. Second, not all parts of a diagram are interesting at all times. Third, users may employ a specific semantic filtering to decide the important elements of a diagram. Such a filtering could, e.g., be that only the name of a class is shown, or that modelling is restricted to the part of an application related to the user interface.

Collaboration. It has been a striking fact in our user studies that all collaborative construction of models has been co-ordinated as turn-taking. This is somewhat in contrast to other observations on shared drawing [3], but we believe it to be general for the kind of work that object-oriented modelling is about.

What was, however, not co-ordinated via turn-taking, was verbal communication and the use of other artefacts. The people engaged in the meetings, e.g., discussed among themselves while another person was drawing at the whiteboard, or they used other artefacts concurrently.

4.5 Implications for Tool Support for Modelling

The user studies show that computerised support for collaboration and communication in modelling is beneficial. This includes support for turn-taking and not hindering communication. Moreover, context switches in turn-taking need to be fast and transparent to users. Also, possible tool support needs to integrate with a computational environment, i.e., provide functionality such as editing of diagrams, code generation, and reverse engineering. Integration of this functionality should not hinder collaboration.

Aspects of the UML notation were too restraining for initial modelling. A tool may try to help in several ways, including supporting semi-formal, incomplete drawings and integrated informal “freehand” annotations. Moreover, a formal notation is often not completely adequate for the problem at hand. Thus, it should be possible to tweak the notation on the fly, in such a way that the notation becomes more appropriate for the problem, while still preserving the original properties of the notation.

Filtering is needed in a wide sense. A CASE tool provides a potentially unlimited workspace, whiteboards do not. CASE tools often provide filtering mechanisms such as zooming, panning, and showing/hiding attributes on diagrams. On a whiteboard, on the other hand, it is possible to, e.g., contract several relationships into a single relationship. Both kinds of filtering, visual and semantic, are useful and should be integrated.

5 The KNIGHT Tool

The user studies and the study of other CASE tools have been used as a basis for implementing a tool supporting collaborative modelling and implementation. This tool, the *Knight tool*, uses a large touch-sensitive electronic whiteboard (currently a SMART Board, `http://www.smarttech.com`, see Fig. 1) as input and output device. This naturally enables collaboration via turn-taking among developers and users. The interaction with and functionality of the tool is discussed in the next sections.

5.1 Functionality and User-Interface

A major design goal of the Knight tool was to make the interaction with the tool similar to that on an ordinary whiteboard. Therefore, the user interface (Fig. 3) is very simple: it is a plain white surface, where users draw UML diagrams using non-marking pens.

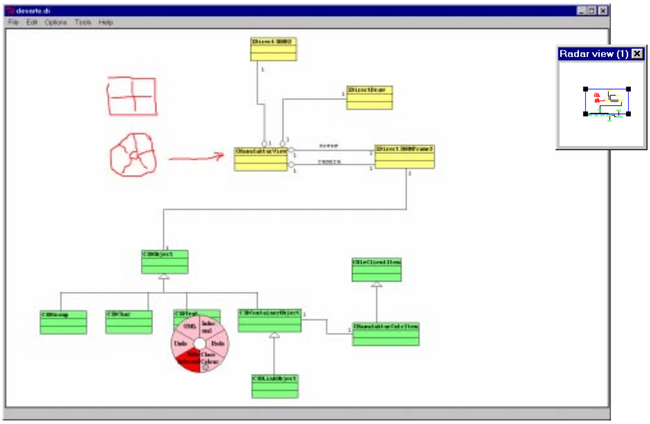


Fig. 3. Knight user interface

The interface is based on gesture input. For example, in order to create a new class, the user simply sketches a rectangle, which the tool then interprets as a class (Fig. 4). Gestures have several advantages over existing toolbars: They allow input directly on the workspace, are fast to draw, and have a low cognitive overhead.

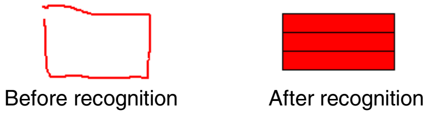


Fig. 4. Recognition of the gesture for a class

Also using gestures, a user may associate two classes by drawing a straight line between them. In general, the gestures for creating UML elements have been chosen so as to resemble what developers draw on ordinary whiteboards. This directness makes the gestures easier to learn and use.

Another way of achieving an intuitive interaction is by using compound gestures [17] and eager recognition [30]. Compound gestures combine gestures that are either close in time or space to one drawing element. So for example, when users draw an inheritance relationship on an ordinary whiteboard, they normally draw a line and then an overlaying triangle. Analogously, in the Knight tool, a user first draws a line between two classes and then a triangle at the appropriate line end.

With eager recognition, the tool continuously tries to classify gestures while they are being drawn. This is used for moving: To move an element, the user draws a *squiggle* gesture on the item to be moved. When the squiggle gesture can be classified with a high confidence, feedback is given in order to show that the gesture was recognised: the item follows the pen.

The technical aspects of the gesture recognition are discussed in the “Design & Implementation” section.

Informality vs. formality. A continuum from informality to formality is supported in two ways. First, users may draw incomplete diagrams, such as relationships only belonging to one class (Fig. 5). The incomplete elements can later be “completed”, e.g., by attaching another class to the relationship.

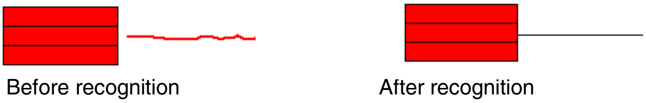


Fig. 5. A relationship with only one class specified

Second, a separate *freehand* mode is provided. In freehand mode, the pen strokes are not interpreted. Instead, they are simply transferred directly to the drawing surface. This allows users to make arbitrary sketches and annotations as on an ordinary whiteboard (see Fig. 3 upper-left). Unlike on whiteboards, these can easily be moved around, hidden, or deleted. Each freehand session creates a connected drawing element that can be manipulated as a single whole.

Navigation. The tool provides a potentially infinite workspace. This allows users to draw very large models, but is potentially problematic in terms of navigating.

Generally there is a need for an easy way of navigating from one point in the diagram to another point in the diagram, and it is desirable to be able to focus on a smaller part of the diagram while preserving the awareness of the whole context. To achieve this in the Knight tool, any number of floating radar windows may be opened (Fig. 6). These radar windows, which may be placed anywhere, show the whole drawing workspace, with a small rectangle indicating the part currently visible. Clicking and dragging the rectangle pans while dragging the handles of the rectangle zooms.

Ordinary pull-down menus are not appropriate for activating the functionality of the tool given the large size of the whiteboard screen. Instead we use gestures as explained above and pop-up menus that can be opened anywhere on the workspace. The pop-up menus are implemented as *pie menus* that are opened when the pen is pressed down for a short while (see Fig. 7). For faster operation, the commands can also be invoked by drawing a short line in the direction of the pie-slice holding the desired command [16]. Furthermore, the menus are context-dependent. The left side of Fig. 7 shows the default menu, whereas the right side shows a more specialised menu that is opened when close to the end of a relationship.

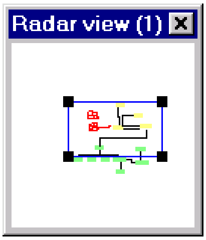


Fig. 6. Radar windows provide context awareness

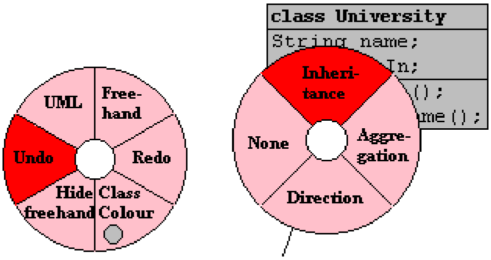


Fig. 7. Context-dependent pie menus

Inputting text. The tool offers five different ways of inputting text (Fig. 8). Apart from normal keyboard input these are: *Virtual keyboard* (a), *Stylus-based Gestures* (as on PDA's) (b), *Cirrin* (c) [20], and *Quikwrite* (d) [27]. They are shown in increasing order with respect to speed of text-entry and difficulty to learn. There is thus support for both casual users without any knowledge of the more specialised ways of inputting text, and advanced users who wish to enter text quickly.

5.2 Design & Implementation

The Knight tool is implemented in the Itcl [22] object-oriented extension of Tcl/Tk [25]. Since the implementation uses Microsoft COM [29] for tool integration, it currently only runs on the Microsoft Windows platform.

Software Architecture. The software architecture of the Knight tool is shown in Fig. 9 using a UML package diagram. The Knight tool and CASE tools that Knight is integrated with are separate processes. The connectors between these processes are, as discussed further below, currently implemented using Microsoft COM.

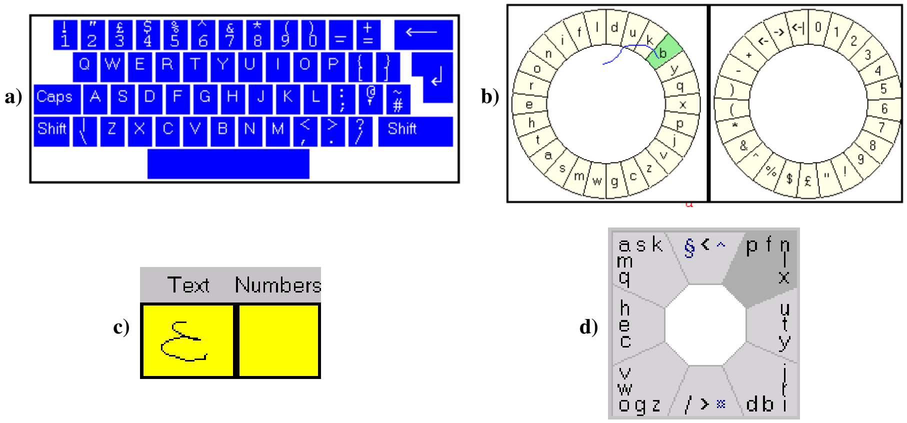


Fig. 8. Text input possibilities in Knight

Internally in Knight, connectors are either event broadcasts or direct method calls. The structure of Knight follows the *Repository* architectural pattern [32]. The packages surrounding the Repository are *readers* and *readers/writers*. They work independently on the repository.

The *Repository* is a repository of UML diagrams. Basically, the class structure is a subset of the UML metamodel. When changes occur in the diagram, Observers [10] on the diagram are notified.

The *Radar* is observing the Repository and it gives an overview by reading and representing the state found in the diagrams.

The *Workspace* both reads and writes from and to the diagrams to display and edit these. Writing in the Repository is implemented using a combination of the Composite and the Command patterns [10], providing undo/redo functionality.

The *CASE Tool Integrator* is a Mediator [10] between the Knight tool and multiple CASE tools (see below).

Gesture Recognition. Rubine's algorithm [30] is used for gesture recognition. The main advantage of this algorithm is that it is relatively easy to train: For each gesture to be recognised, it must simply be provided with a number of prototypical examples of the gesture. Based on these examples, the algorithm then computes a representative vector of features. Examples of features are the total length of a gesture and the total angle traversed by a gesture. Subsequently, these representative vectors can be compared to a vector computed from a user's input, and the most resembling gesture can be chosen based on a statistical analysis.

A few circumstances complicate the gesture recognition a little. First, different types of input devices have different physical characteristics. Thus, the recogniser should ideally be trained once per type of input device. Second, Rubine's algorithm requires that different types of gestures are distinguishable with respect to the feature vector. We handle this by using compound gestures, e.g., when drawing different types of relationships, thus reducing the number of gestures to be recognised. Third, all users do not draw, e.g., a rectangle starting in the same corner and in the same direction. In order to preserve the intuitiveness of and the resemblance to ordinary whiteboards, the gesture recogniser must thus be able to recognise a rectangle starting in all four corners going both clockwise and counter-clockwise. To lessen the burden

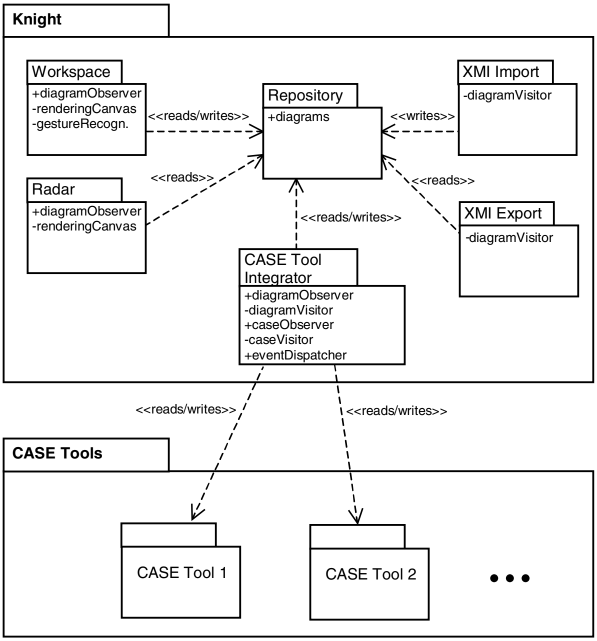


Fig. 9. Software architecture of the Knight tool

of training the recogniser with all these variations, we have implemented a script that takes, e.g., a set of rectangle examples all drawn starting in one corner and going in one direction and then permutes these by rotating and mirroring to obtain eight combinations. In this way the full gesture set of 49 different gestures can be achieved from only 9 sets of examples.

Integration with other CASE tools. There is a need to provide common CASE tool functionality such as code generation and reverse engineering. Different tools support different tasks well and users choose to use different kinds of tools based on the work at hand. Thus, we have chosen an integration strategy instead of building a full-featured CASE tool.

We are implementing both batch and incremental integration with CASE tools. Batch integration is currently achieved in two ways: using *XML Metadata Interchange* format (XMI [24]) and using component technology. To implement the XMI exchange, we used a DTD based on the UML metamodel. Given this DTD, we implemented support for importing models from and exporting models to XML files following its grammar. Since the DTD is generated as specified in the XMI standard, it enables us to share models with other CASE tools that support UML and XMI.

Incremental integration is done for two reasons. First, batch integration may lose information and rely on other tools' interpretation of information. Second, we wish to integrate with CASE tools that support incremental round-trip engineering [4]. For this to be useful, incremental updates are needed so that code is always available and changes in code are immediately reflected in diagrams. The incremental integration is currently done via Microsoft COM [29].

The details of the integration are described in a companion paper [7].

6 Evaluation and Future Work

6.1 Evaluation

We have performed qualitative evaluations of the Knight tool in use. The primary objective of the evaluations has been to evaluate Knight in real work settings. Typically, a facilitator introduced the tool briefly. Following this, each subject was given a chance to try the interaction of Knight and learn the gestures for manipulating diagrams. The facilitator also helped if the subjects had problems using the tool in their work.

After the introduction, the subjects used Knight to work on their current project. While the subjects worked on their project, the use was videotaped and notes were taken. After the sessions, the subjects were interviewed.

The evaluations were all positive and showed that the tool was useful in the design situation. Also, the subjects considered the tool to be a better enabler for both collaboration and creativity than traditional CASE tools. They especially liked the large electronic whiteboard's collaboration support and the tool's interaction style, with its combination of informal and formal elements.

A number of minor problems surfaced. A few of the gestures were hard to learn by some of the participants, although they seemed to learn them as they used the tool further. Also the integration of freehand, formal, and informal elements was generally construed as useful (see Fig. 10), but further functionality was desired, e.g., to associate a freehand element and a formal element to each other.

In summary, the evaluations validate our general approach to supporting collaborative modelling. The issues raised as results of the evaluations were mostly smaller interaction problems, and these were outweighed by the tool's advantages. The users especially found that the tool's lighter, more intuitive interaction allowed for more creativity and that the tool less frequently than ordinary CASE tools caused breakdowns in the modelling.

6.2 Future Work

Longitudinal Studies of Actual Use. The tool should be evaluated in real settings over a longer period of time, preferably for the duration of a whole project. Further evaluation could also include a comparative, quantitative study of the Knight tool's impact on the quality of the resulting design and the amount of time spent using the tool.

Distributed Use. Considering distributed use of the Knight tool suggests some interesting possibilities. The obvious possibility is to connect two electronic whiteboards in different locations and to have designers work on the same model synchronously. But other combinations may also be possible. For example, one might choose to have a different view on a model via an ordinary PC located in the same meeting room as the electronic whiteboard.

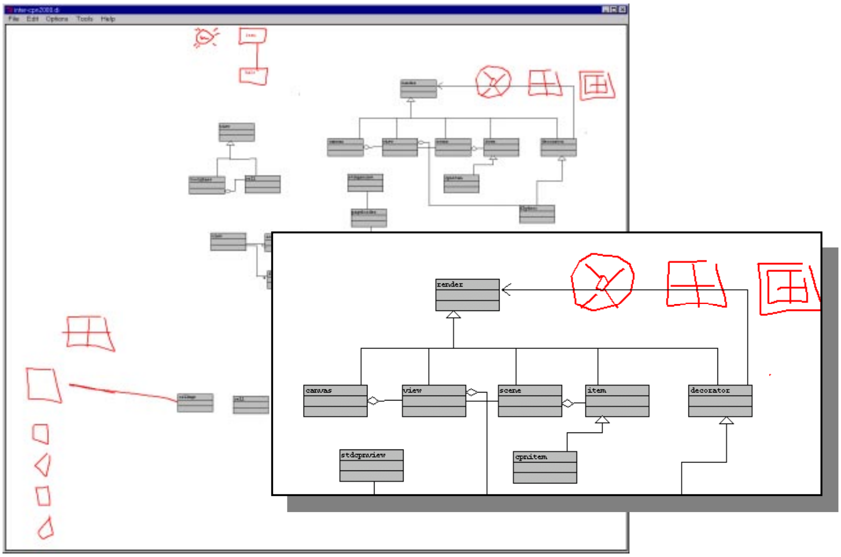


Fig. 10. Diagram produced during an evaluation session (with part of the diagram emphasised)

End User Customisation. An area of research that we so far have left more or less untouched is the ability for the user to customise the tool in a number of ways. Currently, we are working with the idea of a *personal pen* that holds several settings for the user. In this way, a user who chooses a specific pen will use the settings of the pen, such as gesture set, freehand/UML mode, or current colour, when interacting with the electronic whiteboard. A bit further along the way is the possibility for the user to customise the notation, ultimately "on the fly". Simple customisations of this type include changing the appearance of drawings or adding user-defined stereotypes to diagrams. A more elaborate tailoring scheme in which the user actively models on the metamodel of the tool may be possible. Such a tool could be considered a light-weight meta-CASE tool [8][9].

Generalisation. Obviously, full support for all the diagram types of the UML is desirable. Moreover, many of the observations that we have made of object-oriented modelling seem to be true of other kinds of formal modelling as well. This suggests that the Knight tool could support other kinds of diagramming with the same basic interaction.

Other generalisations are also possible: Even though the gesture based interaction has been designed for use on an electronic whiteboard, the interaction seems direct and natural and it would be worthwhile investigating its use on an ordinary PC with other input devices, e.g., tablets, mice, or trackballs. We would also like to explore other more exotic input/output devices such as tablets with built in LCD displays, connected PDAs communicating with a large screen, or electronic extensions of traditional whiteboards such as Mimio [33].

7 Conclusion

Support for modelling is one of the major advantages of object-oriented development. Through the conceptual framework underlying object-orientation, both the problem domain of the system and the system to be built can be understood and formulated.

It is often advantageous to have tool support for modelling. Tool support can, e.g., aid in creating and editing models, in checking the syntax and semantics of models and in generating code from the models. Studies have shown, however, that existing *CASE tools* are rarely used. We believe that part of the explanation for this lies in the poor support for intuition, flexibility, and collaboration in CASE tools. To experiment with and to prove the advantages of such support, we have designed and implemented a tool, Knight, which complements existing CASE tools. The Knight tool provides a direct and fluid interaction that in many ways resemble the interaction with regular whiteboards, it provides support for collaboration in its large shared workspace, and it allows for more flexibility by supporting models with both informal and incomplete elements.

The tool has so far been successfully evaluated in qualitative experiments that validated the basic design. Further evaluation and studies of use are to be performed, but we believe that an extension of current CASE tools with support for creativity, flexibility and collaboration as found in Knight can ultimately help in improving the overall quality of development projects.

Acknowledgements. We thank Ole Lehrmann Madsen for comments that improved this paper. This project has been partly sponsored by the Centre for Object Technology (COT, `http://www.cit.dk/COT`), which is a joint research project between Danish industry and universities. COT is sponsored by The Danish National Centre for IT Research (CIT, `http://www.cit.dk`), the Danish Ministry of Industry and University of Aarhus.

References

1. Aaen, I., Siltanen, A., Sørensen, C., & Tahvanainen, V.-P. (1992). A Tale of two Countries: CASE Experiences and Expectations. In Kendall, K.E., Lyytinen, K., & DeGross, J. (Eds.), *The impact of Computer Supported Technologies on Information Systems Development* (pp 61-93). IFIP Transactions A (Computer Science and Technology), A-8.
2. Blomberg, J., Suchman, L., & Trigg, R. (1994). Reflections on a Work-Oriented Design Project. In *Proceedings of PDC '94*, pp. 99–109, Chapel Hill, North Carolina: ACM Press.
3. Bly, S.A. & Minneman, S.L. (1990). Commune: A Shared Drawing Surface. In *Proceedings of the Conference on Office Information Systems* (pp. 184-192). ACM Press.
4. Christensen, M. & Sandvad, E. (1996). Integrated Tool support for design and implementation. In *Proceedings of the Nordic Workshop on Programming Environment Research*, Aalborg, May 29-31.
5. Christensen, M., Crabtree, A., Damm, C.H., Hansen, K.M., Madsen, O.L., Marqvardsen, P., Mogensen, P., Sandvad, E., Sloth, L., & Thomsen, M. (1998). The M.A.D. Experience: Multiperspective Application Development in Evolutionary Prototyping. In *Proceedings of ECOOP'98*, Bruxelles, Belgium, July, Springer-Verlag, LNCS series, volume 1445.

6. Damm, C.H., Hansen, K.M., & Thomsen, M. (2000). Tool Support for Cooperative Object-Oriented Design: Gesture Based Modeling on an Electronic Whiteboard. In *Proceedings of Computer Human Interaction (CHI'2000).* Haag, The Netherlands, 2000.
7. Damm, C.H., Hansen, K.M., Thomsen, M., & Tyrsted, M. (2000). Tool Integration: Experiences and Issues in Using XMI and Component Technology. In *Proceedings of TOOLS Europe'2000.* Brittany, France.
8. Ebert, J., Süttenbach, S. & Uhe, I. (1997). Meta-CASE in Practice: a Case for KOGGE. In Olive, A. & Pastor, J. A. (Eds.): *Advanced Information Systems Engineering, Proceedings of the 9th International Conference, CAiSE'97*, pp. 203-216, Barcelona, Catalonia, Spain, June 16-20, LNCS 1250, Berlin: Springer.
9. Englebert, V. & Hainaut, J.-L. (1999). DB-MAIN. A Next Generation Meta-CASE. In *Information Systems*, pp. 99-112, 24 (2).
10. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object/Oriented Software.* Addison-Wesley.
11. Greenbaum, J. & Kyng, M. (1991). *Design at Work: Cooperative Design of Computer Systems*. Hillsdale New Jersey: Lawrence Erlbaum Associates.
12. Hughes, J., King, V., Rodden, T., & Andersen, H. (1994). Moving Out of the Control Room: Ethnography in System Design. In *Proceedings of CSCW'94* (pp. 429-439). Chapel Hill: ACM Press.
13. Iivari, J. (1996). Why Are CASE Tools Not Used? In *Communications of the ACM*, 39(10).
14. Jarzabek, S. & Huang, R. (1998) The Case for User-Centered CASE Tools. In *Communications of the ACM*, 41(8).
15. Kemerer, C.F. (1992). How the Learning Curve Affects CASE Tool Adoption. In *IEEE Software,* 9(3).
16. Kurtenbach, G. (1993). *The Design and Evaluation of Marking Menus.* Unpublished Ph.D. Thesis, University of Toronto.
17. Landay, J.A. & Myers, B.A. (1995). Interactive Sketching for the Early Stages of User Interface Design. In *Proceedings of CHI'95*, 45-50.
18. Lending, D. & Chervany, N.L. (1998). The Use of CASE Tools. In Agarwal, R. (Eds.), *Proceedings of the 1998 ACM SIGCPR Conference*, ACM.
19. Madsen, O.L., Møller-Pedersen, B., & Nygaard, K. (1993). *Object-Oriented Programming in the BETA Programming Language*, ACM Press, Addison Wesley.
20. Mankoff, J. & Abowd, G.D. (1998). Cirrin: A Word-Level Unistroke Keyboard for Pen Input. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST).*
21. Martin, J. & Odell, J.J. (1998). *Object-Oriented Methods. A Foundation.* Second Edition. Prentice Hall.
22. McLennan, M.J. (1993). [incr Tcl]: Object-Oriented Programming. In *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11.
23. Moran, T.P., Chiu, P., Harrison, S., Kurtenbach, G., Minneman, S., & van Melle, W. (1996). Evolutionary Engagement in an Ongoing Collaborative Work Process: A Case Study. In *Proceedings of CSCW'96,* 150-159.
24. Object Managment Group (1998). *XML Metadata Interchange* (XMI), document ad/98-07-01, July.
25. Ousterhout, J. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.
26. Pedersen, E.R., McCall, K., Moran, T.P., & Halasz, F.G. (1993). Tivoli: An Electronic Whiteboard for Informal Workgroup Meetings. In Proceedings of *INTERCHI'93*, 391-398.
27. Perlin, K. (1998). Quikwriting: Continuous Stylus-Based Text Entry. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST)..*

28. Reiss, S.P. (1996). Software Tools and Environments. In *ACM Computing Surveys*, 28(1), CRC Press, March 1996.
29. Rogerson, D. (1997). *Inside COM. Microsoft's Component Object Model*. Microsoft Press.
30. Rubine, D. (1991). Specifying Gestures by Example. In *Proceedings of SIGGRAPH'91*, 329-337.
31. Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Addison Wesley.
32. Shaw, M. (1996). Some Patterns for Software Architectures. In Vlissides, Coplien, Kerth (Eds.), *Patterns Languages of Program Design 2.* Addison Wesley, 1996.
33. Yates, C. (1999). Mimio: A Whiteboard without the Board. In *PC Computing*, June 28.

Design Patterns Application in UML

Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel

IRISA/CNRS, Campus de Beaulieu, F-35042 Rennes Cedex, FRANCE
email: sunye,aleguenn,jezequel@irisa.fr

Abstract. The Unified Modeling Language (UML) currently proposes a mechanism to model recurrent design structures: the parameterized collaborations. The main goal of this mechanism is to model the structure of Design Patterns. This is an interesting feature because it can help designers to point out pattern application without spending time with intricate design details. Moreover, it can also help designers to better document their systems and to manage their own design pattern library, which could be used in different systems or projects. However, from a tool perspective, the semantics associated to parameterized collaborations is still vague. To put it more precisely, the underlying representation of a design pattern and of its application, and the binding between these two levels is not exactly defined, and therefore, can be interpreted in different ways. This article has two purposes. First, we point out ambiguities and clarify some misunderstanding points concerning parameterized collaborations in the "official" UML literature. We also show the limits of this mechanism when effectively modeling design patterns. Second, we propose some workarounds for these limits and describe how a tool integrating this mechanism could help with the semi-automatic application of design patterns.

1 Introduction

Design patterns [11] integration into a modeling language is a tempting idea. A simple modeling construct allowing to explicitly point out participant classes of a design pattern could help designers in many ways. Besides the direct advantage of a better documentation and the consequent better understandability of a model, pointing out the existence of a design pattern allows designers to abstract known design details (e.g. associations, methods) and concentrate on more important tasks.

Another tempting idea, consequent to the first one, is to provide tool support to this modeling language and therefore, to design patterns. The automatic implementation of patterns can help overcome some adversity encountered by programmers [27] [4] [10]. More precisely, a tool can ensure that pattern constraints are respected (e.g. that a subject always notifies its observers when it is modified), avoid some implementation burden (e.g. creating several forwarding methods in the Composite pattern) and even recognize pattern within source code, avoiding them to get lost after their implementation.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 44–62, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The UML community succumbed to the first temptation: the latest version of the Unified Modeling Language [25] has improved the *collaboration* design construct in order to provide better support for design patterns. Indeed, the two conceptual levels provided by collaborations (i.e. parameterized collaboration and collaboration usage) fit perfectly to model design patterns. At the general level, a parameterized collaboration is able to represent the structure of the solution proposed by a pattern, which is enounced in generic terms. The application of this solution i.e. the terminology and structure specification into a particular context (so called instance or occurrence of a pattern) can be represented by collaboration usages.

However, UML parameterized collaborations suffer from a lack of precision, which constrains the effective benefits of tool support and makes the second temptation less seductive.

Section 2 is dedicated to the representation of design patterns in UML, and tackle a number of ambiguities that hinder tool support: Pattern occurrences are presented in Sect. 2.1, and it is explained how the occurrences of a pattern are linked to its general description in a precise way. The general description of patterns is supported in UML through the notion of parameterized collaborations, and Sect. 2.2 is dedicated to the syntactic and semantic issues arising from the use of collaborations in the context of design pattern. The limits of UML collaborations are carefully analyzed in Sect. 2.3. Ideas to overcome the shortcomings of collaborations are sketched in Sect.2.4, providing some guide-lines to model the "essence" of design patterns more accurately.

Once the most important issues pertaining to the presentation of design patterns in UML have been pointed out, better modeling of design pattern becomes possible. Effective tool support for UML design patterns is proposed in Sect. 3. Section 3.1 presents the main features that a user is likely to expect from an effective design pattern tool. Relying on the transformation framework provided by the UMLAUT prototype (presented in Sect. 3.2), we show in Sect. 3.3 how a metaprogramming approach allows for powerful manipulations of design patterns, easing the task of designers significantly.

2 Design Patterns and UML Collaborations

2.1 Representing Occurrences of Design Pattern

Had it not provided some support for the notion of pattern, it would have been hardly possible for UML to sustain its role as a unifying notation for object-oriented modeling. Therefore the abundant documentation on UML has sections wholly dedicated to patterns. Figure 1 presents an example of what represents an occurrence of the Composite design pattern, as given in the UML Reference Guide [26]:

The UML 1.3 notation for occurrences of design patterns is in the form of a dashed ellipse connected by dashed lines to the classes that participate in the pattern.

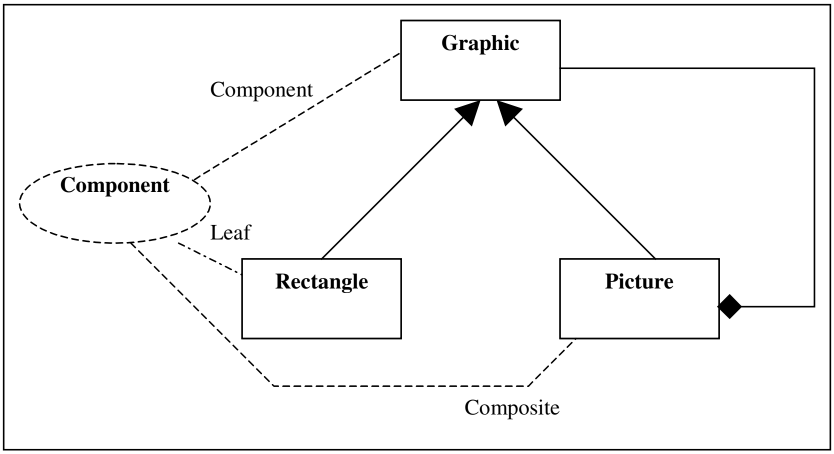


Fig. 1. An occurrence of the Composite Design Pattern

The interpretation of this model seems obvious, even though the Composite design pattern has been (erroneously) renamed as "Component": a Picture is composed of a set of Graphics (Rectangles or other Pictures). Objects instances of class Rectangle can only play the role of a Leaf, and hence cannot contain other Graphics. An experienced designer will also understand that Picture implements methods that will forward every message it receives to its components. In the context of this pattern occurrence, adding another class (say Circle) that also plays the Leaf role looks simple: Figure 28-4 in the UML User Manual [3] shows an example of a similar pattern occurrence, where a same role is assigned to different classes, by simply using as many dashed lines with the same label as necessary.

However, a small remark on that same page explains that "these [two] classes are bound a little differently than the others". This is an evidence that the apparent simplicity actually hides a lot of complex representation issues (see section 2.3 for a detailed discussion). Of course, the user should be shielded from those details, but the UML tool designer is not.

Note that Fig. 1 does not explain how implementation trade-offs are set and what the benefits of representing an occurrence of a design pattern are, other than for documentation purposes. Actually, UML does not support the representation of implementation trade-offs. The designer has no other choice but using comments to point these out.

Of course, the designer can reuse the pattern occurrence symbol for a given pattern any number of times, with a different binding for each new context in which the pattern appears (UML User Manual [3], p.388).

So far we have only seen how a pattern occurrence is made explicit in a model thanks to the dashed ellipse symbol. Two fundamental issues still remain to be solved:

1. Specifying how a pattern occurrence refers to the corresponding pattern specification. That is, we should give a precise meaning to dashed ellipses and dashed lines.
2. Specifying the pattern itself as formally as possible in UML. The UML actually provides a mechanism for this purpose, based on collaborations and genericity. It also provides a mapping of dashed ellipses and lines in this context. However, we will see in section 2.2 that this mechanism entails some confusion and suffers from many shortcomings.

2.2 The Official UML Proposal: Parameterized Collaborations

Design patterns are supposed to be modeled using parameterized collaborations, which are rendered in the UML in a way similar to template classes (UML User Manual [3], p.384). According to [3] p.387, three steps are needed to model a design pattern:

1. Identify the common solution to the common problem and reify it as a mechanism;
2. Model the mechanism as a collaboration, i.e. a namespace containing its structural, as well as its behavioral aspects;
3. Identify the elements of the design pattern that must be bound to elements in a specific context and render them as parameters of the collaboration;

The last two steps give an idea of how a design pattern is supposed to be modeled (and how a collaboration editor might work).

A collaboration is defined in terms of *roles*. The structural aspects of a collaboration are specified using ClassifierRoles, which are *placeholders* for objects that will interact to achieve the collaboration's goal. As a placeholder, a role is similar to a free variable or to a formal parameter of a routine. It will later be bound to an object that *conforms* to the ClassifierRole. Several objects can play one given role at run-time (constraints on the actual number are specified by the multiplicity of the classifier role) and each of them must conform to the classifier role. ClassifierRoles are connected by AssociationRoles, which are placeholders for associations among objects.

The way conformance of an object to a specific role is defined is particularly interesting, and this is where the notion of *base* of a role intervenes. A ClassifierRole does not specify exhaustively *all* the operations and attributes that an object conforming to this role must have. Only the features strictly necessary to the realization of the role are specified as features available in the ClassifierRole. Therefore, the ClassifierRole can be seen as a restriction (or projection) of a conforming object's "full" Classifier to the needed subset of features. Actually, UML imposes that roles be defined *only* as a restriction of existing classifiers and there are OCL rules in the meta-model (see [25] p.2-108) that enforce this view. The classifier(s) that a ClassifierRole is a restriction of is called the *base(s)* of the role.

An object is said to conform to a particular role if it provides all the features needed to play this role, that is, all the features declared in the ClassifierRole.

Although this is not strictly required (as explained in [25] p.2-113) any object that is an instance of a role's base classifier will by definition conform to this role.

On page 199 of [26], the authors propose a model for the Composite design pattern [11] (cf Fig. 2). This model is composed of three roles (Component, Leaf and Composite). The reusability of a design pattern is expressed using the genericity mechanism: to make the above description of Composite context-independent and hence reusable in the context of geometric shapes, the base classifiers of each role are turned into template parameters of the collaboration. Putting the pattern in context simply consists in *binding* the template collaboration to the user model by providing actual arguments for template parameters. This is the official mapping given in [25] for the dashed ellipse symbol in Fig. 1, the actual arguments being inferred by the dashed lines. The label on each dashed line leaving the ellipse corresponds to the name of the role the actual argument will be a base of. With the conformance as defined above, the binding ensures that any instance of the Graphic classifier can play the role of a Component in the collaboration resulting from the template's instantiation represented in Fig. 1.

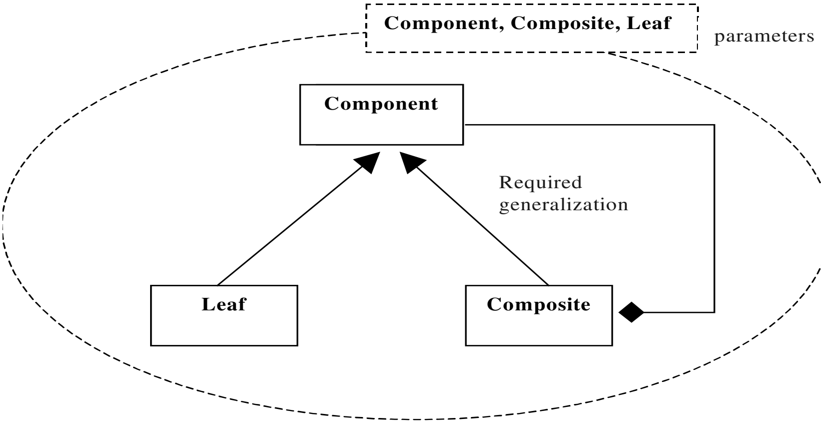


Fig. 2. Composite Design Pattern as a Parameterized Collaboration

The parameterized collaborations cannot however be bound with just any actual classes. The participant classes must respect the *fundamental constraints* of the pattern. Several kinds of constraints can be imposed by the collaboration using a set of *constraining elements*:

- A generalization between two (formal generic) base classifiers means that a similar inheritance relation must also exist between the two corresponding actual classifiers used in any binding of the collaboration. This is just the

UML version of *constraint genericity*. For example, the classes acting as bases for the role Composite and Leaf must be specializations of the class acting as base for the role Component. The inheritance relationship between actual classifiers imposed by such a constraint need not be direct, though.

- An association among two (formal generic) base classifiers means that a similar association must also exist between the two corresponding actual classifiers used in any binding of the collaboration. Such associations among base classifiers are very likely to act as bases for association roles within the collaboration.

2.3 Limitations of Parameterized Collaborations

However, there are severe limitations to the expressive power of constraining elements associated to a collaboration:

Constraints on Generalizations: The graphical representation of collaboration superimposes classifier roles with their corresponding bases, which makes the use of generalization arrows ambiguous. Contrary to generalization relationships between their respective bases, generalization relationships between the classifiers roles themselves do not bring supplementary constraints: they just mean (as usual) that the child role inherits all features of its parent(s) (see [25] p2-105). This graphical ambiguity is the root of confusing descriptions in the UML books themselves [26]. Consider the following quotes:

> In a parameterized collaboration (a pattern) some of the classifier roles may be parameters. A generalization between two parameterized classifier roles indicates that any classifiers that are bound to the roles must satisfy the generalization relationship.
>
> (...) for the common case of parameterized roles, the template is bound by specifying a class for each role.

This can lead the readers to think that the template parameters are the classifier roles, when only base classifiers of the roles actually are template parameters (if a role were a template parameter, then the corresponding actual argument of the binding should also be a role, not simply a classifier. This is possible only if the binding is nested within a broader collaboration that will provide the actual role).

Constraints on Associations: The UML documentation (see e.g. [26]) does not consider it useful to make the bases of association roles template parameters of the collaboration. It is assumed that the bases of association roles can be deduced automatically from the existing associations among the base classifiers given when the template collaboration is bound. However, there are cases when this assumption does not hold: when there are several candidate associations, or when there is no (direct) association. The former case forces an arbitrary choice, while the latter case requires the creation of a <<derived>> association to provide the necessary shortcut through available (indirect) associations. For instance, in the Observer pattern, there could be a mediator object between an observee and an observer, and the association between observee and observer would be computed by going through the mediator. The way an association can be derived (or computed) from existing but indirect ones is explained in [25] p2-19.

Constraints on Available Features: To take part in the collaboration, a role must dispose of a set of available features, which it gets from its base(s) classifier(s) (see [25] p2-108). But if the base is itself a generic parameter, where do the available features come from? Although this problem has not been raised so far in the UML literature, we can envision several possibilities:

1. Defining a special-purpose classifier associated with the template collaboration. This classifier would hold the features needed by a given role, and a constraining generalization relationship would ensure that the actual classifier used as a base for this role is a subclass of this special class. But then, since a ClassifierRole is a special kind of Classifier, why not instead simply define the needed features within the ClassifierRole itself and get rid of the base altogether? (see more on this alternative solution later on).
2. Defining *compound template parameters* and associated conformance rules for the corresponding actual argument in a binding. The available features would then come from *within* the template parameter. The number of so-called well-formedness rules involved in defining the conformance to such template parameters would probably be dissuasive.
3. Turning all needed features into template parameters (the UML does not impose any constraints on the kind of entity allowed as template parameters). We would then need a set of supplementary Constraints to ensure that the actual arguments for the features are actually owned by the appropriate actual arguments for the base classifiers. Constraints written in the Object Constraint Language (OCL [29]) would be added in the set of constraining elements of the collaboration (a Constraint with an upper case 'C' is a kind of UML modeling element, and is represented by an expression enclosed in curly braces.) Note that such a constraint needs access to the meta-level, which is not normally accessible from user-level models.

Constraints Involving any Number of Classifiers: In section 2.1, we have already recognized the need to somehow link several classes of the user model to a single role (or base thereof) of a design pattern. For example, one might want to show that several classes like Rectangle, Circle, Square, etc. all represent Leaves of the Composite design pattern in the context of geometric shapes. Figure 28-4 in the UML User Manual [3] similarly shows that several classes of the model are Commands of the Command design pattern.

Unfortunately, this is not as obvious as one can think. Indeed, the normal mapping of the pattern occurrence symbol is to bind exactly one actual class to exactly one template parameter (which is the base for a role). With bindings as defined in UML 1.3, it does not make any sense to provide several actual arguments for one template parameter. Moreover, templates have a fixed number of template parameters. As already mentioned in section 2.1, [3] avoids the problem by saying that the binding is different in this case: the dashed lines are supposed to map to generalization links which automatically turn all actual arguments into subclasses of a common class (respectively named Leaf and Command) provided by the collaboration itself. This exception to the normal mapping rule is rather confusing: note how the Leaf and Command classes are still represented in the upper-right corner of the collaboration symbol, as if they were still template parameters.

We argue that this proposed solution is more an ad-hoc workaround than a generalizable principle: it is applicable only because adding new *classes* during the binding for the Leaf or the Command roles does not really add any supplementary constraints on the structure of the design pattern solution. The only new information provided is that new kinds of *objects* can play these roles in the collaborations at run-time. It is remarkable that all examples of design patterns that we found in the UML literature are patterns for which this workaround is applicable.

But let us think about modeling the Visitor design pattern [11] in UML, an occurrence of which is given in Fig. 4. A parameterized collaboration will not be sufficient to represent “the” general solution, because the number of template parameters representing the nodes of the structure to be visited is frozen. But this number of nodes impacts the whole structure of the pattern, since the Visitor class must have the right number of accept operations.

We cannot address this problem with a simple workaround. We need to express constraints on the pattern that require full reflexive capability (if only to check the number of operations for example). Giving OCL expressions access to the meta-level (which is not normally accessible from user models) is a promising way to resolve the issue. The standard <<metaclass>> and <<powertype>> UML stereotypes might also prove useful as hooks into the meta-level.

Temporal or Behavioral Constraints: The mere existence of necessary operations or attributes for playing a role is of course not enough. Patterns also prescribe how operation calls, updates of attributes and other actions are organized to achieve a particular task. For instance, in the context of the Observer Design Pattern, any modification of the observee’s state must eventually be followed by a call to `notify()`. UML Interactions can be attached to a collaboration to specify its behavioral aspects. An interaction is a partial order on messages and actions pertaining to the goal, and can be graphically represented as a sequence diagram. Such sequence diagrams often accompany the description of patterns by Gamma et al. in [11].

Being part of the parameterized collaboration, interactions are involved in the binding process, although the UML does not define how. There are two ways interactions can potentially affect the binding:

1. The interaction in the template might be transposed as-is in the resulting model, with template parameters substituted with actual arguments of the binding. This kind of macro expansion is the way UML genericity is supposed to work (see [25] p.2-26).
2. A more sensible approach is to consider the interactions as a new kind of constraint that must be respected by the actual arguments. Of course, the actual participants in the binding might satisfy the constraints as a more or less direct consequence of their own behavioral organization. For instance, a completely unrelated operation call might be inserted between two calls prescribed by the pattern. This of course should not invalidate the pattern usage, since such a situation is bound to happen in all but trivial situations.

It should now be clear that interactions attached to collaborations ought to be interpreted as behavioral constraints on participants, not unlike temporal logic formulas. Satisfaction of these constraints or formulas should be part of the conformance rules of pattern bindings, but UML 1.3 does not provide many hints on this issue. Some recent work [20] on formalizing UML collaborations and interactions shares this view of conformance of objects to collaboration roles.

To emphasize the formula-like nature of these constraints, an interesting approach worth investigating is to turn them into textual constraints written in a variant of OCL extended with temporal logic operators. Such an extension to OCL is proposed in [21].

The notion of base is ad hoc We have seen that the notion of base classifier of a role was at the heart of design pattern definition in UML, since roles are defined with respect to their base, and reusability is provided by making these bases formal parameters of the collaboration which then becomes a generic template.

We have also just seen that this use of bases as template parameters seriously complicates the way roles are specified in terms of available features, and suffers from the inherent limitations of template instantiations (e.g., fixed number of parameters in a UML Binding).

These problems suggest that the notion of base is probably not the right way to relate roles and classifiers of objects that will conform to the roles. Making roles dependent on pre-existing, "external", base classifiers when they could have stood on their own, looks suspicious. These dependencies impair the reusability of the collaborations, since they can't be reused without the corresponding bases, and they require clairvoyance on the part of the designer if several design patterns are to be combined (they might need shared bases).

The dependency between roles and classifiers should actually be in the opposite direction, as it is very likely that the classifier of an object will be designed by first carefully considering all the roles that the object may play in the various collaborations of the system. Hence the classifier of this object will be obtained by *merging* the roles' specifications, not the other way round.

The UML notion of *Realization* would be more appropriate than the notion of *Binding* of template parameters to express the fact that a given classifier realizes a set of roles. Moreover, realizations can specify a *mapping* expression to describe precisely how the classifier should realize the roles. We are investigating how the transformation functions proposed in section 3.3 would fit in this context.

2.4 Constraints as the Essence of Patterns

On the one hand, the various limitations listed above make it impossible for UML parameterized collaborations to precisely specify some of the more interesting constraints of design patterns. Their expressive power is limited to the prescription of associations and generalization links, and to a certain extent, the availability of features. Undoubtedly, more sophisticated constraints need access to the UML meta-model.

On the other hand, the static structure of collaborations (and of associated interactions if they are not considered as constraints) entails many choices that are not fundamental to a pattern itself, but are specific of some of its reified solutions. This prevents UML collaborations from representing only the *essence* of a pattern, free of any premature choices. All diagrams representing patterns or pattern occurrences in the UML literature fall short of this ambitious goal because of the over-specification side-effect of collaborations.

The essence of a pattern is what Eden [8] calls a "leitmotiv", that is, the intrinsic properties of a pattern *and nothing more.* These properties are common to all variants of a given pattern reification. Therefore, they should be expressed as general constraints over such reifications, which presumably involves meta-level OCL-like expressions and temporal logics.

How an enhanced OCL would fit together with UML collaborations and genericity to fully represent the essence of a pattern is at the heart of accurate UML specifications of patterns, and is still an active research topics of the authors.

A transformation tool would ideally provide this level of pattern modeling, and would help the user solve the corresponding constraints (or offer transformations implementing them, with meta-programs). The next section further elaborate on this idea of sophisticated tool support for UML design patterns.

3 Towards a UML Pattern Implementation Tool

The goal behind the above study of parameterized collaborations (and their limits) is to provide effective support for design patterns in a UML tool. But before extending the description of automatic design patterns implementation, let us dispel some possible misunderstanding concerning the integration of pattern in a CASE tool. According to James Coplien [7] p. 30 - *patterns should not, can not and will not replace programmers* - , our goal is not to replace programmers nor designers but to support them. We are not attempting to detect the need of a design pattern application but to help designers to explicitly manifest this need and therefore abstract intricate details. We are also not trying to discover which implementation variant is the most adequate to a particular situation, but to discharge programmers from the implementation of recurrent trivial operations (e.g. message forwarding) introduced by design patterns.

Consequently, our goal is to propose a tool that allows designers to model the structure of design patterns, to explicitly identify the participant classes of a pattern application and to map the structure of a pattern into any application of this pattern. Once this is possible, the tool can automate different approaches of patterns use. According to Florijn et al. [10], a pattern-based tool can follow three main approaches:

- Recognition: In this approach, the tool recognizes that a set of classes, methods and attributes corresponds to a design pattern application and points this out to the designer;
- Generation: Here, the designer chooses a pattern she wants to apply, the participant classes and some implementation trade-off and receives the corresponding source code;
- Reconstruction: The former approaches can be merged into this third approach. Here, the tool modifies a set of classes that looks like a pattern application into an effective pattern application. This modification implies the addition or modification of classes, attributes and methods.

Since our goal is to integrate design patterns into a UML tool and therefore adopt a generative approach, our present interest concerns only the last two approaches.

The idea behind the second approach is that the pattern solution could be seen as a sequence of steps. Therefore, a pattern implementation would be obtained if the tool follows these steps. Actually, the implementation is more complicated than that since the tool should verify if the pattern has not already been partially implemented. Moreover, the solution is not unique, the implementation may change according to certain trade-offs.

The third approach is very close to Opdyke's refactorings [19], i.e. operations that modify the source code of an application without changing its behavior. One of the many difficulties of this approach is that design patterns specify a set of solutions to a problem, but do not (or rarely do) specify a common situation to which the pattern should be applied. This seems to be reasonable, since the problem described by a pattern can be solved in several other manners and it would be impossible to catalog each manner.

Another approach, which was not enumerated by Florijn et al. concerns design patterns validation. More precisely, design patterns have implicit constraints that could be automatically verified. For instance, in the Observer pattern, a tool could verify if every method that modifies the subject also notifies its observers.

In our perspective, a design tool that support design patterns should, on one hand, provide high level transformations that help designers to apply a design pattern and, on the other hand, ensure that the applied solution remains consistent during the design process. In the next sections, we will further explain what exactly we want our tool to do.

3.1 A Pattern Tool in Action

In order to further describe the rationale behind our pattern tool, let us take the role of a designer using her favorite UML CASE tool. More specifically, she is in the middle of the design of her application and her model contains many classes and operations.

Suppose that she now decides to apply a specific design pattern to her model. This could be done in two different ways.

The first way is to select a class and choose the transformation she wants to apply to this class. In this case, the designer knows exactly what she wants to do and uses the pattern application as a macro. This transformation could be, for instance, the creation of a Composite class (i.e. the application of the Composite pattern). The tool will automatically create a new class, an association between the new class and the selected one and a set of forwarding methods in the new class. At the same time, the tool will create an usage of the Composite collaboration (defined previously) and assign the composite role to the new class and the component role to the selected class. This approach will be further explained in section 3.3.

The second way consists in creating an usage of a collaboration, which corresponds to the pattern she wants to use. In this case, she has already applied (partially or not) this pattern to her classes and wants to document it. She will manually determinate which classes participate to this pattern occurrence. Then, the tool will try to automatically bind non-specified participants (classes, features, associations, etc.) to the collaboration, and ask for the designer validation.

From then on, all the constraints inherent to the chosen pattern should be continuously checked as a background process. Any unsatisfied constraint should be clearly indicated: For each of them, an item would appear in a "to-do list", describing the action the user should take in order to make the pattern application complete and correct. To the extent that it is possible, the tool should propose some semi-automatic steps, to relief the user as much as possible. As soon as all constraints are satisfied, the user can proceed with code generation to obtain the final application.

3.2 UMLAUT's Transformation Framework

UMLAUT is a freely available tool dedicated to the manipulation of UML models. UMLAUT notably provides a UML transformation framework allowing complex manipulations to be applied to a UML model [13]. These manipulations are expressed as algebraic compositions of elementary transformations.

We propose the use of a mix of object-oriented and functional paradigm to develop a reusable toolbox of transformation operators. The general approach consists of two major steps. The first phase uses an iterator to traverse the UML meta-model instance into a linear sequence of model elements. The second phase maps a set of operators onto each element in this sequence.

In the context of the theory of lists, it has been shown that any operation can be expressed as the algebraic composition of a small number of polymorphic operations like *map*, *filter* and *reduce* [2].

The transformation process can be generalized into three stages: element selection, element processing and reductive validation. We can re-apply the first two stages repeatedly using composition of *map* and *filter* to achieve the desired results.

3.3 A Metaprogramming Approach

This metaprogramming approach consists in applying design patterns by means of successive transformation steps that should be applied starting from an initial situation up until a final situation is reached where the occurrence of the pattern is explicit. For instance, Fig. 3 presents a situation to which the Visitor pattern can be applied, i.e. a class hierarchy where several methods (optimize() and generate()) are defined by every class. Applying the pattern to this hierarchy means creating another class hierarchy to where these methods will be transferred. The final situation is presented in Fig. 4.

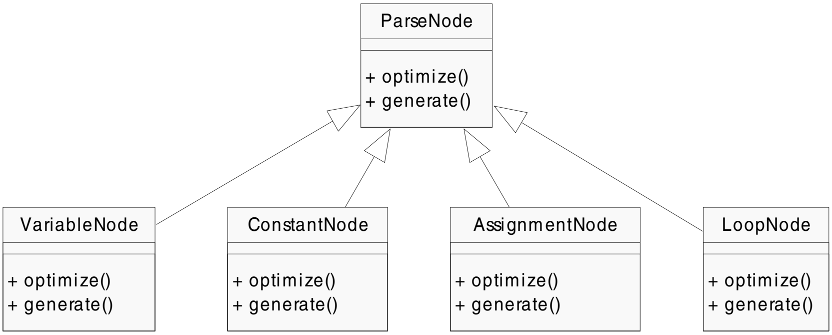


Fig. 3. The Visitor Design Pattern - Initial state

One may accurately notice that this transformation approach is not valid for every design pattern, since only a few patterns (e.g. Bridge, Proxy) mention an existing situation to which the pattern should be applied. This is true, this approach is not and does not intend to be universal. Our prime intent here is to provide UML designers meta-programming facilities that we believe every software designer should have [24].

The above reference to Smalltalk refactorings is not naive; we are strongly convinced that every development tool should have such facility and that this facility can help developers to apply design patterns. However, refactorings cannot be directly translated to UML for two reasons. First, refactorings were defined for programming languages and UML has several modeling constructs other than

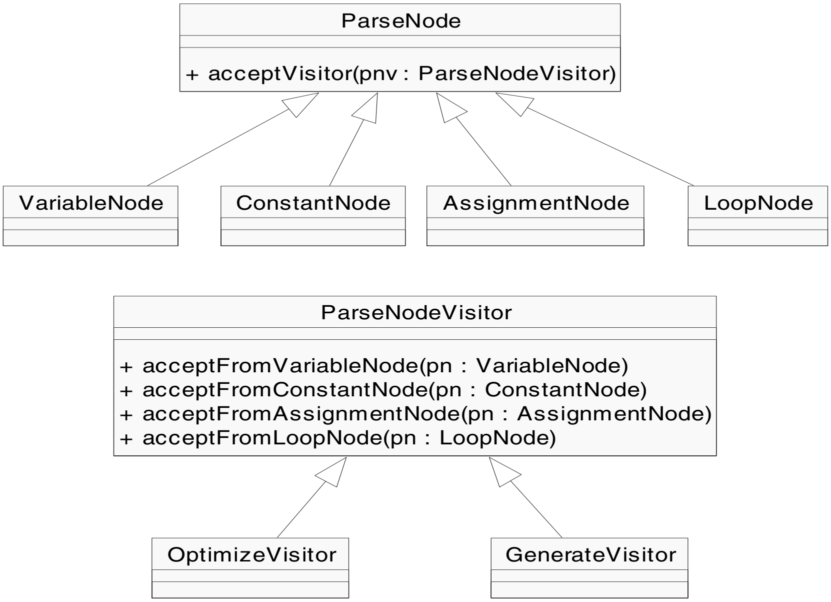

ParseNode
+ acceptVisitor(pnv : ParseNodeVisitor)
VariableNode
ConstantNode
AssignmentNode
LoopNode
ParseNodeVisitor
+ acceptFromVariableNode(pn : VariableNode)
+ acceptFromConstantNode(pn : ConstantNode)
+ acceptFromAssignmentNode(pn : AssignmentNode)
+ acceptFromLoopNode(pn : LoopNode)
OptimizeVisitor
GenerateVisitor

Fig. 4. The Visitor Design Pattern - Final state

classes, methods and variables. Second, the behavior of a UML method is described by a few diagrams and constraints whilst refactorings operate directly over Smalltalk code. Hence, some refactorings should be changed and new ones must be defined. Nevertheless, there is at least one advantage of using UML: OCL [29] can easily be used to define refactorings pre and post conditions. For instance, the addClass() refactoring [22] can be defined as:

```
Package::addClass(newClass:Name, superclasses,
                                  subclasses:Set(Class)):Class
pre:
not self.allClasses()->exists(name=newClass) and
subclasses->forAll(each | each.allSuperTypes()->
    includesAll(superclasses))
post:
result.name = newClass
self.allClasses()->includes(result) and
self.classReferences(result)->isEmpty and
result.allSuperTypes()->intersection(superclasses)->size() =
    result.allSuperTypes()->size() and
result.allSubTypes()->intersection(subclasses)->size() =
    result.allSubTypes()->size()
```

This function is interpreted as follows: a new class can be added to a package if this package does not already have a class named as the new one and if the future subclasses of the new class are subclasses of its future super-classes. After the class addition, the package will contain a class named as the new class, there will be no references to the new class and the list of the new class super-classes will be exactly the same as the super-class list provided as a parameter.

We intend to allow designers to combine multiple refactorings in order to represent design pattern applications. UMLAUT currently provides a transformation framework [13], where UML model transformations can be specified. However, since this framework cannot be specialized at run-time (some compilation is needed), we expect to use a syntax close to OCL to describe refactorings and combine them.

To illustrate a transformation function, we present below a function which specifies the application of the Visitor pattern. This example was copied, with a few changes, from [23]. These changes were necessary since the original example applies to Smalltalk code whilst here it applies to UML models:

```
Class::addVisitor()
actions:
let abstractVisitor :=
     self.package.addClass(self.name+'Visitor', nil, nil).
self.allOperations()->forAll(operation |
     let concreteVisitor := self.package.addClass(
               operation.name+'Visitor', abstractVisitor, nil).
     self.allSubTypes()->forAll(subclass |
          subclass.allOperations()->select(subop |
               subop.hasSameSignature(operation)->forAll(op |
                    op.move(concreteVisitor,'acceptFrom'+subclass.name).
                    op.rename('acceptVisitor').
                    op.pullUp()))))).
```

This transformation applies to the class that is to play the role of element in the Visitor pattern. Its application to the ParseNode class operates as follows. At first, an abstract visitor class named ParseNodeVisitor is created. Then, for each method of the ParseNode class, the transformation will:

1. Create a concrete visitor, subclass of the abstract visitor;
2. Move all same signature methods (from subclasses) to this concrete visitor. This operation replaces the body of the original method by a forwarding method that simply calls the new method. The moved method is renamed and receives a new parameter allowing it to refer to the members of its original class.

A natural extension to this transformation would be the addition of a collaboration usage, specifying that ParseNode and ParseNodeVisitor are participant classes of an occurrence of the Visitor pattern.

4 Related Work

The rapid evolution of design patterns has been followed by several research efforts on pattern-based tool development. These efforts pursue different goals, such as: design pattern recognition [15] [5]); formal specification [18] [16] [1]; code reuse [27] [17]; and code generation [6] [28].

PatternWizard is one of the most extensive projects of design patterns specification, and has influenced our research work in several points. PatternWizard proposes LePUS [9] a declarative, higher order language, designed to represent the generic solution, or *leitmotif* , indicated by design patterns. In order to define the constructs of this language, the authors have analyzed the solution of all Gamma et al. design patterns and identified a set of common building block, called *tricks* (an extensive list of tricks is given by Amnon H. Eden in his PhD thesis [8]).

Tricks specify a sequence of operations over an abstract syntax language. Tricks are divided in three levels: Idioms, Micropatterns and Design Patterns. Idioms are the first level of tricks and operate over the abstract syntax language. They abstract language dependence. Tricks can be compared to refactorings in many ways, except in code generation: in opposition to refactorings, tricks can generate code. Micropatterns are the second level of tricks (they can be defined as a set of idioms). They represent simple mechanisms that appear repeatedly among design patterns such as, for instance, message forwarding. Finally, design patterns are the higher level of tricks and represent the *leitmotif* of design patterns.

The structure of design patterns can be more precisely defined by LePUS declarations, or formulae, than a simple class diagram. Indeed, a LePUS formula can define the multiplicity of each participant class in the design pattern application. Furthermore, the behavior of participant methods is precisely defined by tricks.

Our work differs from PatternWizard in two aspects. First, we use UML and OCL to specify patterns. We believe that a UML collaboration and OCL rules can be more intelligible than the LePUS formulae and its associated graphical language. Second, PatternWizard works at code level and is not integrated to any design model.

Such integration is proposed by Kim et Benner [14]. They propose to split design into two levels (both described in OMT). The *pattern* level is situated above the *design* level. Design level is composed by classes, their components and associations that represent together the result of design. Above this level, the pattern level defines additional semantics. The main idea is to link design constructs which participate in a pattern application to the structure representing the model itself. This rationale is very close to UML parameterized collaborations. However, unlike collaborations, a certain flexibility exists (and is defined). Pattern occurrences need not be totally isomorphic, they respect the concept of generalizable path, that takes into account the generalization links present in the design model.

Design and implementation integration is also provided by Fred [12], a development tool designed for framework development and specialization. Fred helps developers to specialize application frameworks, indicating hot-spots and inviting them to follow a sequence of steps, presented by a *working-list*. Doing this, Fred can reduce the time necessary to effectively use a framework. In Fred, developers can explicitly precise an occurrence of a pattern using links between the pattern description and participant classes of this occurrence. After this, Fred proposes a set of template methods (also presented by a working-list) that should be completed to fully implement the pattern. Template methods describe constraints concerning the behavior of participant classes.

Finally, Jan Bosch proposes LayOM [4], a layered object model. LayOM intends to integrate design patterns, using an extended object model that supports the concept of layers. More specifically, the layers encapsulate a set of objects that intercept and treat sent messages. LayOM integrates two more concepts, category and state. A category is an expression that describes the characteristics of a set of possible clients, that should be treated likewise. A state is the abstraction of the internal state of an object. Message interception, provided by layers are appropriate to the implementation of some patterns, such as Adapter, for instance. LayOM generates C++ code.

5 Conclusion

The extensive study of the solutions proposed throughout the UML literature evidences many ambiguities in the use of parameterized collaborations and a lack of semantic foundation preventing systematic analysis and manipulation of design patterns in UML. We still consider however that sophisticated support for design patterns in a UML tool is not out of reach.

The knowledge that we acquired throughout the former development of a pattern implementation tool [28] and the implementation of the full UML meta-model in UMLAUT was extremely valuable when we started working on tool support for design patterns in UML. It helped us point out the crucial difficulties paving the way of automatic implementation of design pattern.

Getting a good grasp of the complexity underlying UML parameterized collaborations is definitely not a simple task, especially when they are used in the context of design patterns. We promptly searched for solutions in the abundant UML literature, but did not get complete, unambiguous, authoritative answers. Section 2 brings new insights on this topic, and also raises several issues for which there is no satisfying solution yet.

Since parameterized collaborations are not totally adapted to model design patterns, we now face a delicate choice between adapting the present semantics of collaborations or extending the UML meta-model with new constructs to bridge current semantic gaps. We are investigating this latter approach. in order to provide a more accurate way of binding Design Patterns and their occurrences, and a better support for implementation trade-offs and feature roles. In addition, we are specifying a set of UML specific refactorings and implementing them using our transformation framework.

In spite of our efforts, some questions are still left open:

First, keeping the binding between a general pattern description and the participants in a particular occurrence up-to-date might be problematic if the user is allowed to dynamically change the variant applied. Indeed, the set of participants may also have to be changed, and so will the set of actual constraints to be satisfied.

Second, checking for constraint satisfaction is likely to be largely undecidable, or prohibitively expensive in term of computational resources. Model-checking techniques might however provide useful results to check some behavioral constraints.

Third, automation of the step needed to satisfy a constraint is possible in isolation, but when a modeling element takes part in more than one pattern occurrence, the number of possibilities the tool could suggest quickly becomes unmanageable.

At the present time, a working version of UMLAUT is freely available for download[1]. This version provides some model construction facilities, Eiffel code generation and evaluation of OCL constraints.

References

1. P. S. C. Alencar, D. D. Cowan, and C. J. P. Lucena. A formal approach to architectural design patterns. In J. Woodcock M. C. Gaudel, editor, *FME'96: Industrial Benefit and Advances in Formal Methods*, pages 576–594. Springer-Verlag LNCS 1051, 1996.
2. R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springler-Verlag, 1987.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.
4. Jan Bosch. Language support for design patterns. In *TOOLS Europe'96*, pages 197–210. Prentice-Hall, 1996.
5. K. Brown. Design reverse-engineering and automated design patterns detection in Smalltalk. *Technical Journal*, 1995.
6. F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.
7. James O. Coplien. *Software Patterns.* SIGS Management Briefings. SIGS Books & Multimedia, 1996.
8. Amnom H. Eden. *Precise Specification of Design Patterns and Tool Support in Their Application.* PhD thesis, University of Tel Aviv, 1999.
9. Amnon H. Eden, Amiram Yehudai, and Joseph Gil. Patterns of the agenda. In *LSDF97: Workshop in conjunction with ECOOP'97*, 1997.
10. Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 472–495. Springer-Verlag, New York, N.Y., June 1997.
11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, Reading, MA, 1995.
12. Markku Hakala, Juha Hautamäki, Jyrki Tuomi, Antti Viljamaa, and Jukka Viljamaa. Pattern-oriented framework engineering using FRED. In *ECOOP '98—Workshop reader on Object-Oriented Technology*, Lecture Notes in Computer Science, pages 105–109. Springer-Verlag, 1998.

[1] http://www.irisa.fr/pampa/UMLAUT/

13. Jean-Marc Jézéquel, Wai Ming Ho, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
14. Jung J. Kim and Kevin M. Benner. A design patterns experience: Lessons learned and tool support. In *Position Paper, Workshop on Patterns, ECOOP'95*, Aarhus, Denmark, 1995.
15. C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Working Conference on Reverse Engineering (WCRE'96)*. IEEE CS Press, 1996.
16. Anthony Lauder and Stuart Kent. Precise visual specification of design patterns. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 114–134. Springer, 1998.
17. Marco Meijers. *Tool Support for Object-Oriented Design Patterns*. PhD thesis, Utrecht University, 1996.
18. Tommi Mikkonen. Formalizing design patterns. In *ICSE'98*, pages 115–124. IEEE CS Press, 1998.
19. William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992.
20. Gunnar Övergaard. A formal approach to collaborations in the unified modeling language. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.
21. S Ramakrishnan and J McGregor. Extending OCL to support temporal operators. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems (WM3)*, Los Angeles, California, USA, May 1999. ACM press.
22. Don Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, 1999.
23. Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.
24. Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk - why every Smalltalker should use the Refactoring Browser. *The Smalltalk Report*, 14(10):4–11, 1997.
25. UML RTF. *OMG Unified Modeling Language Specification, Version 1.3, UML RTF proposed final revision*. OMG, June 1999.
26. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
27. Jiri Soukup. Implementing patterns. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 395–412. Addison-Wesley, Reading, MA, 1995.
28. Gerson Sunyé. Génération de code à l'aide de patrons de conception. In Jacques Malenfant and Roger Rousseau, editors, *Langages et Modèles à Objets - LMO'99*, pages 163–178, Villeneuve s/ mer, 1999. Hermes. In French.
29. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

UML-F: A Modeling Language for Object-Oriented Frameworks

Marcus Fontoura[1], Wolfgang Pree[2], and Bernhard Rumpe[3]

[1] Department of Computer Science, Princeton University
35 Olden Street, Princeton, NJ 08544-2087, U.S.A
mfontoura@acm.org
[2] C. Doppler Lab for Software Research, University of Constance
D-78457 Constance, Germany
pree@acm.org
[3] Software and Systems Engineering, Munich University of Technology,
D-80290 Munich, Germany
rumpe@acm.org

Abstract. The paper presents the essential features of a new member of the UML language family that supports working with object-oriented frameworks. This UML extension, called UML-F, allows the explicit representation of framework variation points. The paper discusses some of the relevant aspects of UML-F, which is based on standard UML extension mechanisms. A case study shows how it can be used to assist framework development. A discussion of additional tools for automating framework implementation and instantiation rounds out the paper.

1 Introduction

Object-oriented (OO) frameworks and product line architectures have become popular in the software industry during the 1990s. Numerous frameworks have been developed in industry and academia for various domains, including graphical user interfaces (e.g. Java's Swing and other Java standard libraries, Microsoft's MFC), graph-based editors (HotDraw, Stingray's Objective Views), business applications (IBM's San Francisco), network servers (Java's Jeeves), just to mention a few. When combined with components, frameworks provide the most promising current technology supporting large-scale reuse [16].

A framework is a collection of several fully or partially implemented components with largely predefined cooperation patterns between them. A framework implements the software architecture for a family of applications with similar characteristics [26], which are derived by specialization through application-specific code. Hence, some of the framework components are designed to be replaceable. These components are called variation points or hot-spots [27] of the framework. An application based on such a framework not only reuses its source code, but more important, its architecture design. This amounts to a standardization of the application structure and allows a significant reduction of the size and complexity of the source code that has to be written by developers who adapt a framework.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp.63–82, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Recent standardization efforts of the Unified Modeling Language (UML) [32] offer a chance to harness UML as notational basis for framework development projects. UML is a multi-purpose language with many notational constructs, however, the current standard UML does not provide appropriate constructs to model frameworks. The constructs provided by standard UML are not enough to assist framework development, as will be discussed during the rest of this paper. There is no indication in UML design diagrams what are the variation points and what are their instantiation constraints. Fortunately, UML provides extension mechanisms that allow us to define appropriate labels and markings for the UML model elements.

This paper describes how to explicitly model framework variation points in UML diagrams to describe the allowed structure and behavior of variation points. For this purpose, a number of extensions of standard UML are introduced. The extensions have been defined mainly by applying the UML built-in extensibility mechanisms. These extensions form a basis for a new UML profile [7, 33, 35], especially useful for assisting framework development. This new profile is called UML-F.

The main goal of this paper is to introduce some key elements of UML-F and to demonstrate their usefulness. It would be beyond the scope of this paper to introduce the whole set of UML-F extensions. One of the main goals of defining UML-F was to try to use a small set of extensions that capture the semantics of the most common kinds of variation points in OO frameworks. In this way the designer can profit from his or hers previous experience with UML and learn just a few new constructs to deal with frameworks. This paper describes how the extensions have been defined allowing others extensions that deal with new kinds of variation points to be added to UML-F if needed. The current version of UML-F was refined based on the experiences of a number of projects [11]. These experiences have shown how UML-F can assist the framework development and instantiation activities to reduce development costs and at the same time increase the resulting quality of the delivered products. This paper presents a condensed version of a real-application case study to illustrate the benefits of UML-F and its supporting tools.

The rest of this paper is organized as follows: Section 2 outlines the UML extensions and discusses how they can be used to explicitly represent framework variation points. It also shows how the extensions allow for the development of supporting tools that can assist framework development and instantiation. Section 3 describes a case study of real application of UML-F, illustrating its benefits. Section 4 discusses some related work. Section 5 concludes the paper and sketches our future research directions.

2 The Proposed UML Extensions

This section introduces UML-F through an example. It summarizes the new extensions and presents a general description of their semantics. It also presents a description of the UML extensibility mechanisms and how they have been applied in the definition of UML-F. A description of tools that use UML-F design descriptions to automate framework development and instantiation is also presented.

2.1 Motivating Example

Fig. 1 shows a student subsystem of a web-based education framework [12] in plain UML, where (a) represents a static view of the system (UML class diagram) and (b) provides a dynamic view (UML-like sequence diagram). The dynamic view illustrates the interaction between an instance of each of the two classes.

The *showCourse()* method is the one responsible for controlling the application flow: it calls *selectCourse()*, which allows the student to select the desired course, *tipOfTheDay()*, which shows a start-up tip, and finally *showContent()* to present the content of the selected course.

Method *selectCourse()* is the one responsible for selecting the course the student wants to attend. It is a variation point since it can have different implementations in different web-based applications created within the framework. Different examples of common course selection mechanisms include: requiring a student login, showing the entire list of available courses or just the ones related to the student major, showing a course preview, and so on. There are numerous possibilities that depend on the framework use.

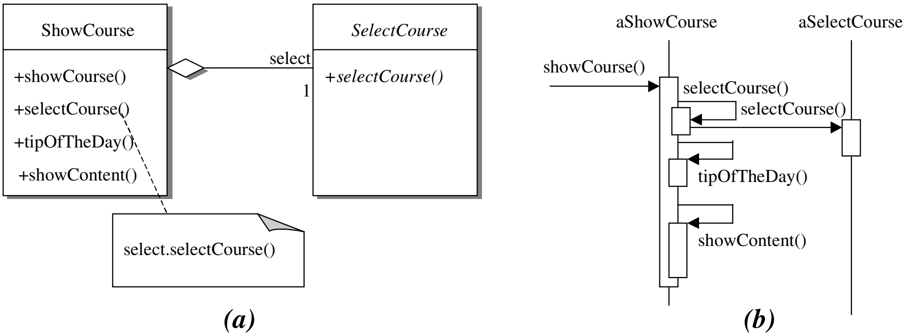


Fig. 1. UML representation of a framework web-based framework.

Fig. 1 shows *selectCourse()* as an abstract method of an abstract class *SelectCourse*. During framework instantiation, the framework users would have to create subclasses of *SelectCourse* and then provide a concrete implementation of the *selectCourse()* method. The problem with this representation is that there is no indication that *selectCourse()* is a variation point in the design diagrams. There is also no indication of how it should be instantiated. Although the name of the abstract method *selectCourse()* is italicized this notation is not an indication of a variation point, rather it indicates an abstract method which does not necessarily have to be a variation point.

Method *tipOfTheDay()* is also a framework variation point. The reason is that some applications created from the framework might want to show tips while others will not do so. The framework should provide only the methods and information that are useful for all the possible instantiated applications and the extra functionality should be provided only in framework instances. Although this may seem a strong statement, it is the ideal situation. The inclusion of methods like *tipOfTheDay()* could lead to a complex interface for *ShowCourse*, with many methods that would not be needed by several framework instances. A good design principle in designing a framework its to try to keep it simple; extra functionality can always be placed in component libraries.

The *Actor* class hierarchy is used to let new types of actors be defined depending on the requirements of a given framework instance. The default actor types are students, teachers, and administrators, however, new types may be needed such as librarians, and secretaries. This means that applications created from the framework always have at least three kinds of actors, students, teachers, and administrators, but several other actor types may be defined depending on the application specific requirements. This design structure is presented in Fig. 2.

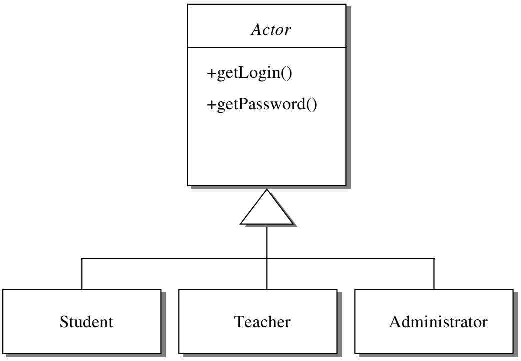


Fig. 2. Actor hierarchy.

The *Actor* class hierarchy also represents a variation point, since it allows the definition of new classes to fulfill the application specific requirements. However, this is not properly indicated in the UML diagram presented in Fig. 2. The framework developer should be able to indicate the variation points in class hierarchies to facilitate the job of the framework user during the instantiation process. Fortunately, UML provides a constraint called *Incomplete* in its standard set of constraints. *Incomplete* indicates that new classes may be added to a given generalization relationship and was adopted as part of UML-F, as will be described in subsection 2.3.

2.2 UML Extensibility Mechanisms

UML provides three language extension mechanisms: stereotypes, tagged values, and constraints. Stereotypes allow the definition of extensions to the UML vocabulary, denoted by *«stereotype-name»*. Each model element (e.g. a class or a relationship) can have a stereotype attached. In this case, its meaning is specialized in a particular way suited for the target architecture or application domain. A number of possible uses of stereotypes have been classified in [2], but stereotypes are still a rather new concept and still subject of ongoing research [7].

Tagged values are used to extend the properties of a modeling element with a certain kind of information. For example, a version number or certain tool specific information may be attached to a modeling element. A tagged value is basically a pair consisting of a name (the tag) and the associated value, written as "*{tag=value}*". Both tag and value are usually strings only, although the value may have a special interpretation, such as numbers or the Boolean values. In case of tags with Boolean values, UML 1.3 allows us to write "*{tag}*" as shortcut for "*{tag=TRUE}*". This leads to the fancy situation that occasionally concepts a stereotype, e.g. *«extensible»*, and a tag, e.g. *{extensible}*, could be used for the same purpose. Since model elements can only have one stereotype, but an unlimited number of tagged values, it is often better to use tagged values in this kind of situation. They provide more flexibility, e.g. freeing us of defining a new stereotype for each combination of tags that may be attached to a model element.

In addition to the mentioned two UML extension mechanisms, there exist constraints. Constraints may be used to detail how a UML element may be treated. However, like the other two, constraints have a rather weak semantics and therefore can be used (and misused) in a powerful way. Constraints are today usually given informally, or by a buzzword only. The *{incomplete}* constraint (Fig. 3) could also be defined as tagged value.

We expect that this mismatch among the extensibility mechanisms be improved in future UML versions. D'Souza, Sane, and Birchenough suggest that all three kinds of extensions should be stereotypes [7]. We argue in favor of this unification, but we will retain the flexibility of tags and therefore will use tagged values for all purposes.

2.3 UML-F Extensions

This subsection introduces UML-F illustrating its application to model the web-based education framework [12]. Fig. 3 models part of the framework representing and classifying the variation points explicitly. The variation points are modeled by a number of tagged values with values of Boolean type to extend the UML class definitions.

In this example the method *selectCourse()* is marked with the tagged value *{variable}* to indicate that its implementation may vary depending on the framework instantiation. The tagged value *{variable}* has the purpose to show the framework user that *selectCourse()* must be implemented with application specific behavior for each framework instance. Methods marked with *{variable}* are referred to as *variable methods*.

In contrast to the previous tagged value, *{extensible}* is applied to classes. In this example *{extensible}* is attached to the *ShowCourse* class, indicating that its interface may be extended during the framework instantiation by adding new functionality, like methods such as *tipOfTheDay()*. Please note that extension is optional, but not a must.

An important point here is that the diagram shown in Fig. 3 is a result of a design activity, and therefore may implemented in several different ways. The fact that a class is marked as *{extensible}* tells us that its implementation will have to allow for the extension of its interface, since a given framework instance may want to do so. However, it does not mean that the new methods have to be added directly to the class. The same holds for variable methods: the changes may be defined without changing the method directly, but by the addition of new classes that provide appropriate implementations for the method. Section 3 discusses some implementation techniques that may be applied to model variable methods and extensible classes.

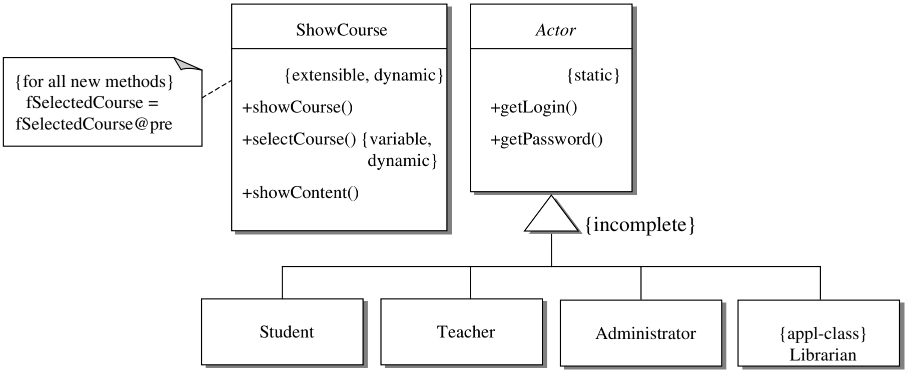


Fig. 3. UML-F extended class diagram.

Fig. 3 uses the tag *{incomplete}* to indicate a third kind of variation point: an *extensible interface. {Incomplete}* is applied to a generalization relationship, allowing new subclasses to be defined by framework instances. In this example it indicates that new subclasses of *Actor* may be provided to fulfill the requirements of applications created from the framework. Please note that *{incomplete}* is already provided by the UML as a constraint, with exactly the same meaning used here.

The tag *{appl-class}* is used to indicate a placeholder in the framework structure where *application specific classes* may be or have already been added. It complements the definition of extensible interfaces: the generalization relationship between an extensible interface and an application class is always *{incomplete}*. Class *Librarian* is an example of an *application class*. The *{incomplete}* tag allows the framework user to create as much application classes from a given extensible interface during framework instantiation as needed. In contrast to the other two kinds of variation points, extensible interfaces have a direct mapping from design to implementation since current OO programming languages provide constructs for modeling generalization relationships directly.

Two other Boolean value tags, called *{dynamic}* and *{static}*, complement the variation point definition by indicating whether runtime instantiation is required. Each variation point can be marked either by the *{dynamic}* or by the *{static}* tag (but not both). Variable methods are instantiated by providing the method implementation. Extensible classes are instantiated by the addition of new methods. Extensible interfaces are instantiated by the creation of new application classes. Interpreted languages, such as Smalltalk and CLOS, give full support for runtime, or *{dynamic}*, instantiation. Java offers dynamic class loading and reflection that also can be used to allow dynamic instantiation of variation points. In the example shown in Fig. 3 the tag *{dynamic}* is used because it is a user requirement to have dynamic reconfiguration for the variation points that deal with course exhibition. The tag *{static}* is used for the *Actor* extensible interface since new actor types do not need to be defined during runtime. The tag *{dynamic}* implies that the implementation has support for runtime instantiation for the marked element. However, such a runtime instantiation must not necessarily happen.

The note attached to the *ShowCourse* extensible class is an OCL [25, 33, 35] formula that defines that the class attribute *fSelectedCourse* shall not be changed by any of the new methods that may be added to the *ShowCourse* extensible class during framework instantiation. This kind of restrictions over variation points is called instantiation restrictions. To be able to describe certain OCL constraints for methods that have neither been introduced nor named yet the tag *{for all new methods}* is used, indicating that this constraint is to hold for all new methods. This kind of tag strongly enhances the power of description of the design language, as it allows us to talk about methods that have not even been named yet.

Although it is beyond the scope of this paper, Fig. 4 shows a sequence diagram that can be used to limit the possible behavior of a variation point. The sequence diagram shows the main interaction pattern for a student selecting a course. As it may be decided by actual implementation, it is optional whether the student has to log in before he selects a course or whether the data is validated. This kind of option can be shown in sequence diagram by using *{optional}* tag, which indicates interactions that are not mandatory. In the area of sequence diagrams, there are many more possibilities to apply tags of this kind for similar purposes, such as determining alternatives, avoidance of interleaving, and so on. We expect useful and systematic sets of tags for sequence diagrams to come up in the near future. Fig. 4 tells us that a concrete method that instantiates *selectCourse()* must have the following behavior:

1. It may display a login web page;
2. It must show a web page for the selection of the desired course;
3. It may validate the data by checking if the login is valid, and whether the student is assigned to the course or not. This step is optional since there can be courses that do not require student identification;

The extended class diagrams and the sequence diagrams complement each other providing a rather useful specification of variation points and their instantiation restrictions. It is important that framework developers provide documentation that describes what parts of the system should be adapted to create a valid framework instances. It is quite cumbersome that framework users today often need to browse the framework code, which generally has complex and large class hierarchies to try to identify the variation points. The diagrams and diagram extensions introduced in this example address this problem. Section 3 will further discuss these ideas, showing how UML-F can assist framework implementation and instantiation.

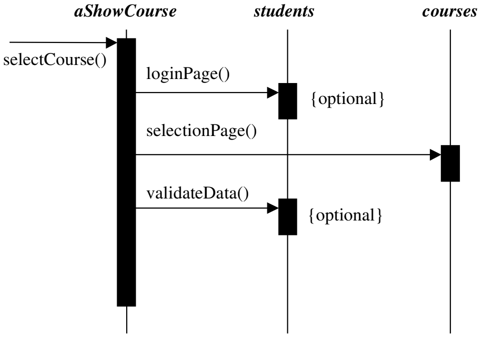


Fig. 4. Sequence diagram for *selectCourse().*

2.4 Language Description

Once the extensions are defined it is crucial to specify their exact meaning. As a side-note, it is important to mention that in most languages (such as natural language, like English), new vocabulary is explained through a definition using existing vocabulary. This even holds for programming languages, like Java, where new classes and methods are defined using existing classes, methods, and basic constructs. Unfortunately, UML 1.3 and high likely also UML 1.4 does not provide a clear path for defining the precise semantics of new stereotypes, tagged values, and constraints. Therefore, this section describes the meanings of our newly introduced elements mainly informally. A formal approach to characterize a variant of these elements based on set theory is presented in [11]. However, this formal definition of UML-F is not presented here since its usefulness for the communication purposes is limited [33].

This paper demonstrates how UML-F deals with three kinds of variation points: *variable methods*, *extensible classes*, and *extensible interfaces*. Variable methods are methods that have a well-defined signature, but whose implementation varies for each instantiated application. In the example *selectCourse()* is a variable method. Extensible classes are classes that may have their interfaces extended during the framework instantiation. *ShowCourse*, for example, may require the addition of new methods (like *tipOfTheDay())* for each different application. Extensible interfaces are interfaces or abstract classes that allow the creation of concrete subclasses during the framework instantiation. The instantiation of this last kind of variation point takes place through the creation of new classes, called *application classes*, which exist only in framework instances.

It should be clear that these three kinds of variation points have different purposes: in variable methods the method implementation varies, in extensible classes the class interface varies, finally, in extensible interfaces the types in the system vary (new application classes may be provided). All three kinds may either be static (do not require runtime instantiation) or dynamic (require runtime instantiation).

There are other kinds of variation points in framework design, such as variation in structure (attribute types for example). Coplien describes several kinds of variability problems in his multi-paradigm design work [6]. They integrate well into UML-F using similar principles to the ones described in this paper. To avoid the explosion of the number of extensions and to keep the presented part of UML-F feasible this paper focus on the most important kinds of variation points.

UML diagrams are extended by the tags *{variable}, {extensible}, {incomplete}, {appl-class}, {static},* and *{dynamic}.* The first two represent variable methods and extensible classes, respectively. *{Static}* and *{dynamic}* are used to classify them regarding to their runtime requirements. The *{incomplete}* tag (in UML 1.3 known as constraint) has been adapted to identify extensible interfaces. The keywords *{extensible}, {variable},* and *{incomplete},* indicate what are the variation points and their exact meaning. The *{appl-class}* stereotype indicates placeholders for classes that are part of instantiated applications only.

OCL specifications [25, 33, 35] may be written on notes as in standard UML, however, they have an enhanced meaning if the notes are attached to variation points. In the case of variable methods, it means that all method implementations that may be defined during instantiation should follow the specification. If an OCL constraint is attached to an extensible class, the special tag *{for all new methods}* is useful to describe the behavior of methods that do not even have a name yet. This tag indicates that the constraint applies to all methods that might be added during instantiation. Similarly, if attached to an extensible interface, the OCL constraint applies to all methods that can be overridden or added to each application class.

Let us also mention the tag *{optional}*. Here, it extends sequence diagrams to indicate that certain interaction patterns are not obliged to occur. These sequence diagrams have proven useful to be applied to all kinds of variation points. Generally, they are used to describe a *pattern behavior* that should be followed by the variation point instances, as shown in Fig. 4. OCL specifications, on the other hand, are generally used to specify invariants that should be satisfied by the variation point instances, as shown in Fig. 3. Thus, sequence diagrams and OCL constraints complement each other in constraining the possible instantiations of variation points, and may therefore be used together.

Table 1 summarizes the new UML-F elements and informally defines their semantics.

Table 1. Summary of the new elements and their meanings

Name of extension	Type of extension	Applies to notational element of UML	Description
{appl-class}	Boolean Tag	Class	Classes that exist only in framework instances. New application classes may be defined during the framework instantiation.
{variable}	Boolean Tag	Method	The method must be implemented during the framework instantiation.
{extensible}	Boolean Tag	Class	The class interface depends on the framework instantiation: new methods may be defined to extend the class functionality.
{static}	Boolean Tag	Extensible Interface, Variable Method, and Extensible Class.	The variation point does not require runtime instantiation. The missing information must be provided at compile time.
{dynamic}	Boolean Tag	Extensible Interface, Variable Method, and Extensible Class.	The variation point requires runtime instantiation. The missing information may be provided only during runtime.
{incomplete}	Boolean Tag	Generalization and Realization	New subclasses may be added in this generalization or realization relationship.
{for all new methods}	Boolean Tag	OCL Constraint	Indicates that the OCL constraint is meant to hold for all newly introduced methods.
{optional}	Boolean Tag	Events	Indicates that a given event is optional. It is useful for specifying a template behavior that should be followed by the instantited variation point.

2.5 Tool Support

This subsection shows how tools that benefit from the UML-F design diagrams may be defined to assist both framework development and instantiation. The tools suggested here have a prototypical implementation using PROLOG. However, many currently available UML case tools give support reasoning about tagged values and could be adapted to work with UML-F. This subsection gives information to allow the customization of UML case tools for working with OO frameworks.

Assisting Framework Development. Standard OO design languages do not provide constructs for representing flexibility and variability requirements. UML-F addresses this problem representing variation points as first-class citizens thus making the framework intentions more explicit. The new language elements are not concerned with how to implement the variability and extensibility aspects of the framework, but focus on representation at design level. Consequently, the diagrams are more abstract (and more concise) than standard OO diagrams. Unfortunately some of the new design elements cannot be directly mapped into existing OO programming languages.

Extensible interfaces can be directly implemented through standard inheritance. Although dynamic extensible interfaces are not supported in compiled languages such as C++, they may be simulated through dynamic linking (Microsoft Windows DLLs, for example). Variable methods and extensible classes, on the other hand, cannot be directly implemented, since standard OO programming languages do not provide appropriate constructs to model them.

To bridge this design-implementation gap, several techniques may be used. Design patterns are a possible solution, since several patterns provide solutions for flexibility and extensibility problems and are based only on extensible interfaces. Thus, design patterns may be used to transform variable methods and extensible classes into extensible interface variation points. Fig. 5 illustrates the use of the Strategy design pattern [15] to implement this mapping. Classes *ShowCourse* and *SelectStrategy* are identified with the tags *{separation, template}* and *{separation, hook}* to indicate the roles they play in the pattern. Strategy is based on the Separation meta-pattern [28], in which a template class is responsible for invoking the variable method in the hook class. The use of tags that indicate meta-pattern roles complement the UML-F description for variation points implemented by design patterns, further clarifying the design. A similar solution for identifying design diagrams with pattern roles is described in [30].

The transformations used to map variable methods and extensible classes into implementation level constructs must be behavior-preserving, since the system functionality is independent of the implementation technique used to model the variation points.

A *code generation* tool can be used to automate design to implementation transformations. It is responsible for mapping the new design elements of UML-F into appropriate implementation level structures. More specifically, it is responsible for eliminating the variable methods and extensible classes from the design. This mapping is based on meta-artifacts that describe the transformations. These meta-artifacts are called implementation models. It is an imperative to allow the definition of new implementation models for variation points, so that different styles of translation are possible.

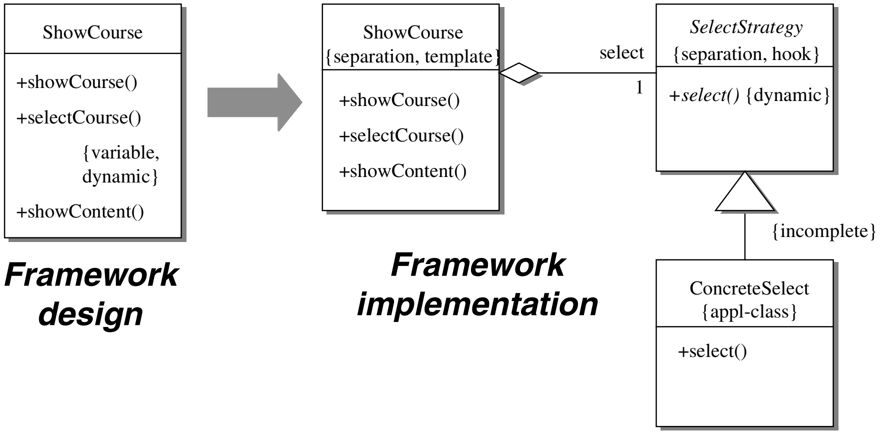


Fig. 5. Transforming variable methods into extension interface variation points.

The transformation illustrated in Fig. 5 is an example of a mapping supported by the code generation tool. The implementation model that supports this transformation describes how dynamic variable methods are modeled by the Strategy design pattern. Fig. 6 illustrates the code for this implementation model, which searches for all variable methods in the design diagrams and applies Strategy to them.

The implementation transformations (illustrated in Fig. 6) preserve the design structure described in *Project* and create *NewProject* to store the generated framework. All the design elements that are not transformed, the kernel elements and the extensible interfaces, are copied from *Project* to *NewProject*. The variable methods and extensible classes are transformed in the way described by the selected implementation model.

```
applyStrategy(Project, NewProject) :-
          [...]
          forall(variableMethod(Project, Class, Method, _),
          strategy(Project, NewProject, Class, Method)),
          [...]

strategy(Project, NewProject, Class, Method) :-
          concat(Method, 'Strategy', NewClass),
          createExtensibleInterface(NewProject, NewClass, dynamic),
          createMethod(NewProject, NewClass, Method, public, none, abstract),
          createAggregation(NewProject, Class, NewClass, strategy),
          [...]
```

Searches for variable methods

Uses strategy to model them

Fig. 6. Strategy implementation model.

Each valid implementation model artifact has to define at least four transformations: (static and dynamic) variable methods and (static and dynamic) extensible classes. Examples of implementation models that have been successfully used to assist framework implementation include different combinations of design patterns, meta-programming [21], aspect-oriented programming (AOP) [20], and subject-oriented programming (SOP) [17], as described in [11]. The case study section also describes some other mappings.

The selection of the most appropriate technique to be used model each variation point is a creative task and cannot be completely automated. However, UML-F diagrams and the set of implementation models available for each kind of variation point may help the framework designer to narrow his or hers search for appropriate implementations. Moreover, the code generation tool automatically applies the transformation once the implementation model has been selected, making the mapping from design to implementation less error prone.

Some UML case tools, such as Rational Rose (http://www.rational.com), allow the customization of how code is generated from the design diagrams. Therefore, it is possible to specify how code should be generated for the new UML-F elements.

Assisting Framework Instantiation. During the framework instantiation, application classes must be provided to complete the definition of the extensible interface variation points (at this point this is the only kind of variation points in the system, given that the other two have already been eliminated during implementation). Fig. 7 illustrates a framework instantiation. After the instantiation all extensible interfaces disappear from the design, since the *{incomplete}* generalizations become "complete." In this example the variation point was instantiated by just one concrete application class, *SimpleSelect,* which is marked by the *{c-hook}* tag to indicate that it plays the role of a concrete hook. In a general case, however, several application classes may be provided for each extensible interface.

An instantiation tool can be used to assist the application developer to create applications from the framework. The tool knows what are the exact procedures to instantiate extensible interfaces: it has to create a new subclass, ask for the implementation of each of the interface methods, and ask for the definition (signature and implementation) for each new method that might be added, if any. The tool prompts the application developer about all the required information to complete the missing information for each variation point in the framework structure.

Note that the tags that indicate the meta pattern roles are useful just for enhancing the design understating, and are not processed by the implementation and instantiation tools.

Depending on the implementation model selected, different instantiation tasks may be required for the same variation point, as will be illustrated in Section 3. UML-F descriptions can be seen as structured cookbooks [22] that precisely inform were application specific code should be added. The instantiation tool is a wizard that assists the execution of these cookbooks. Once again the code generation part of standard UML case tools may be adapted to mark the points in which code should be added by using the information provided by the extensible interface tags.

3 Case Study

This section details the implementation and instantiation of the web-education framework modeled in Fig. 3. It starts from the UML-F specification, derives the final framework implementation, and shows how it may be instantiated. The benefits of UML-F and its supporting tools are discussed throughout the example.

3.1 Framework Implementation

Let us consider that the only variation points of the framework are the ones presented in Fig. 3. Since all the variation points have been identified and marked in the UML-F design diagrams, the next step is to provide implementation solutions to model them. As discussed before, extensible interfaces and the framework kernel (modeled only by standard UML constructs) have straightforward mappings into OO programming languages. Therefore the framework designer focus during the implementation phase should be on how to model variable methods and extensible classes. In this example two variation points have to be examined: the *selectCourse()* variable method and the *ShowCourse* extensible class.

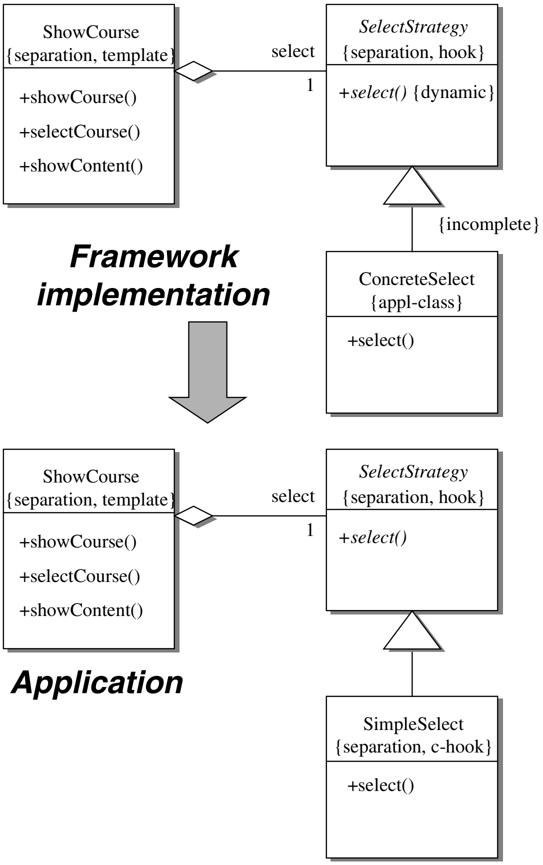


Fig. 7. Instantiation example.

The designer has to select an appropriate technique based on his or hers experience. If a supporting tool with a set of implementation models is available, the analysis of these models may facilitate this task. One of the models available in the code generation tool is the use of the Strategy design pattern [15] to implement dynamic variable methods and a slightly changed version of the Separation meta-pattern [28], which allows the invocation zero or more hook methods, to implement dynamic extensible classes. Since the transformations are automatically applied by the tool let us try this solution and see what happens. The resulting design is shown in Fig. 8.

This solution worked quite well. The solution for extending the *ShowCourse* interface allows the addition of new methods without directly changing the class interface. It allows an instance application to define zero or more methods that will be invoked before the actual content of the course is displayed, and that is the expected behavior. An important point to make is that the instantiation restriction specified by the OCL constraint in Fig. 3 is automatically assured by this solution, since the new methods do not have access to the *fSelectCourse* attribute that is private to *ShowCourse*.

In the case of *selectCourse()*, however, the Strategy solution does not guarantee that the behavior specified by the sequence diagram in Fig. 4 will be followed. Strategy is a white-box pattern since it allows the definition of any behavior for the hook method. The verification of this kind of instantiation restrictions is not an easy task (and is generally an undecidable one), however there are some implementation solutions that may be more restrictive, or more black-box.

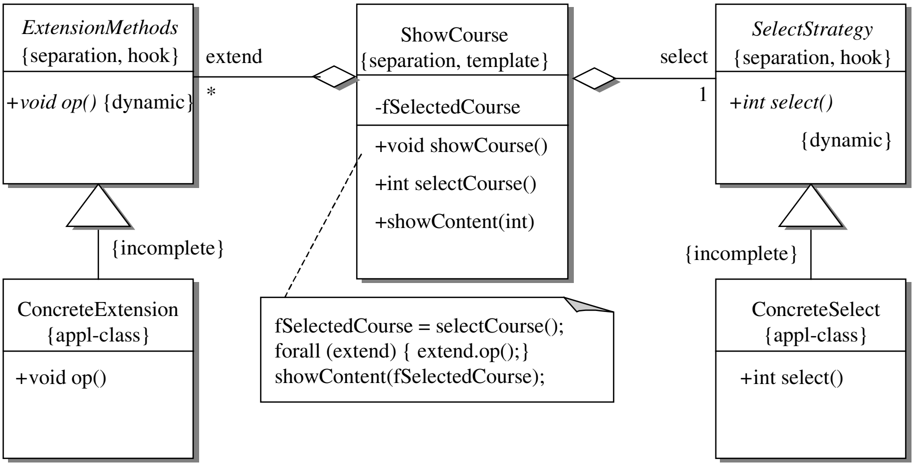


Fig. 8. A pattern-based implementation.

A solution that might be more appropriate for *selectCourse()* is the definition of a meta-object protocol (MOP) [18]. MOPs allow meta-level concepts to be dynamically defined in terms of base-level ones. Thus, the use of MOP may be a good alternative since it is a more restrictive solution than the Strategy pattern: the possible instantiations are just the ones defined by the protocol. Fig. 9 illustrates the use of MOP for this example. Whenever instances of the *SelectMOP* class are created a set of Boolean parameters that complete the variation point behavior have to be provided: *login* (TRUE if login is required), *major* (TRUE if a student can attend only the courses related to his or hers major), and *validate* (TRUE if it is required that the student have to be assigned to be able to attend the course). The combination of these parameters provides all the possible instantiations allowed by the MOP. Note that this solution is much more restrictive than the Strategy solution, but it has the advantage that it always preserves the instantiation restrictions specified in the sequence diagram.

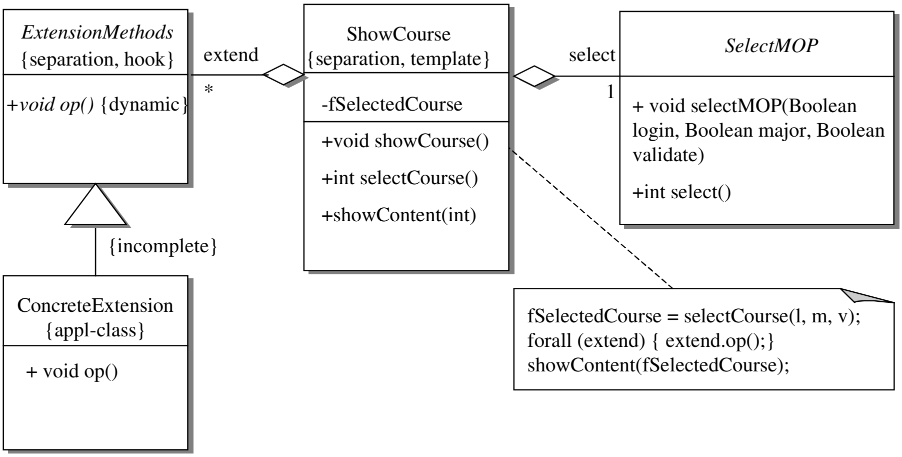


Fig. 9. Using MOP to model selectCourse().

The implementation of MOPs cannot be automated by the code generation tool, since each MOP is specific for a given variation point. However, the UML-F instantiation restrictions provide a good documentation that can be used by the MOP developers. In this example the parameters *login* and *validate* can be directly derived from Fig. 4. In general MOPs may require objets more complex than Boolean ones as parameters and reflection may be required in their implementation.

Note that the runtime constraints *{static}* and *{dynamic}* play a crucial role during framework development. In this example, if the variation points were defined as *{static}* a much simpler design solution based on the Unification meta-pattern [28] could be used for both cases. In Unification-based patterns the template and hook methods belong to the same class, leading to a less flexible but simpler design solution.

3.2 Framework Instantiation

During instantiation the variation points missing information have to be filled with application specific code. Since the variable methods and extensible classes have been eliminated during implementation, only extensible classes are left to be instantiated by the application developers.

Tools such as the instantiation tool may facilitate this task by identifying all the points in which code has to be written. However, even if no tools are available, the UML-F diagrams make this task very straightforward since all the extensible interfaces and their corresponding instantiation restrictions are marked in the diagrams.

Fig. 10 shows an example of application created from the framework defined in Fig. 8. Application classes are provided to complete the definition of the two variation points. Note that if the MOP solution had been adopted the *selectCourse()* variation point would not require new application classes, since MOPs are completely instantiated during runtime by parametrization. This illustrates that different implementation models applied to the same variation point may demand different instantiation procedures.

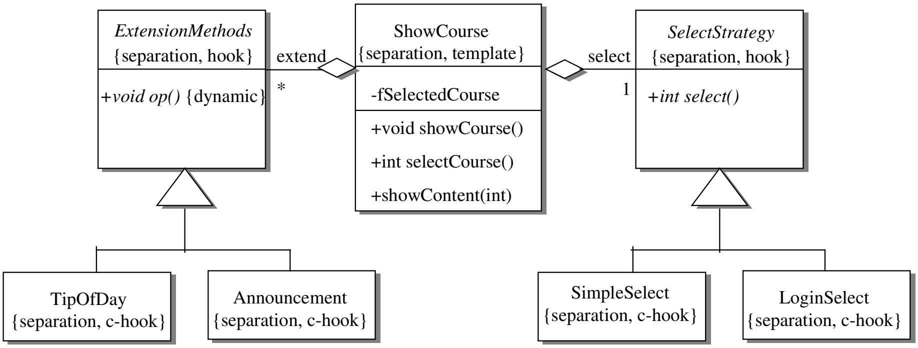


Fig. 10. An application created from the framework.

4 Related Work

This section describes some of the current design techniques used to model frameworks, and relates them to UML-F. It shows that currently proposed constructs used to represent framework variation points have not adequately met our expectations.

Early OO design methods, like OMT [31], as well as the current UML 1.3, provide a number of diagrams for structure, behavior, and interaction. Different OO design notations include different artifacts, such as the representation of object responsibilities as CRC cards [1, 37]. However none of these artifacts has explicit support for the representation of the variation points of a framework.

UML represents design patterns as collaborations (or mechanisms) and provides a way of modeling framework adaptation through the binding stereotype [32]. However, framework instantiation usually is more complex than simply assigning concrete classes to roles: variation points might have interdependencies, might be optional, and so on. Catalysis uses the UML notation and proposes a design method based on frameworks and components [8]. Frameworks are treated in Catalysis as collaborations that allow substitution. However, as discussed in the paper, OO application frameworks may require different instantiation mechanisms. Therefore, Catalysis and standard UML only partly address the problems identified in this paper due to a lack of support for explicit marking variation points and their semantics.

Design patterns [4, 15, 36] are usually described using standard OO diagrams. Since various design patterns provide solutions to variability and extensibility problems [15] they define a common vocabulary to talk about these concepts [36] and may enhance the understanding of framework designs. Sometimes design pattern names are used as part of the class names allowing the framework user to identify variation points through the used names. However, in a typical framework design a single variation point class can participate in various design patterns. Then the approach of using design pattern names as class names becomes obfuscated. One possible solution for this problem is the use of role-based modeling technique, as shown in [30].

Meta-level programming [21], which can be seen as an architectural pattern [4], provides a good design solution for allowing runtime system reconfiguration. Therefore, the use of meta-level programming is a useful technique for modeling variation points that require runtime instantiation, and (with appropriate conventions) it may facilitate the identification of variation points in the framework structure. The case study shown in section 3 has shown that both design patterns and meta-level programming can be used in conjunction with UML-F, during the implementation of variation points.

The use of role diagrams to represent object collaboration is a promising field in OO design research [5]. Riehle and Gross propose an extension of the OOram methodology [29] to facilitate framework design and documentation [30]. His work proposes a solution for an explicit division of the design, highlighting the interaction of the framework with its clients. The use of roles does simplify the modeling of patterns that require several object collaborations and provides a solution for documenting classes that participate in several design patterns. However, no distinction is made between the kernel and variation point elements. This problem is handled using design patterns: if the framework user knows what patterns were used to model each of the variation points he or she can have an intuition on how the framework should be instantiated. On the other hand, if the pattern selections are not explicitly represented, the identification of the variation points becomes again difficult. Another disadvantage of this approach is the solution for modeling unforeseen extensions proposed in [30], which may lead to a very tangled design. Although it can be a good solution it should have a more concise representation at design level. This paper has shown how to use roles to complement the description of variation points implemented by design patterns.

Contracts [18, 19] and adaptable plug-and-play components (APPCs) [24] provide linguistic constructs for implementing collaboration-based (or role-based) diagrams in a straightforward manner. They may be used to implement variation points since they represent instantiation as first-class citizens. However, these concepts are still quite new and their use for implementing frameworks needs further investigation. Also Lieberherr and the researchers of the Demeter Project [24] have developed a set of concepts and tools to help and evaluate OO design that can be used to enhance framework development.

The Hook tool [13, 14] uses an extended version of UML in which the variation point classes are represented in gray. This differentiation between kernel and variation points helps framework design and instantiation, but it does not solve the problem completely. Framework designers still have to provide the solutions for modeling each variation point without any tool support. A good point of this approach is that instantiation constraints are treated as first-class citizens in the definition of hooks.

Several design pattern tools [3, 9, 10, 23] have been proposed to facilitate the definition of design patterns, to allow the incorporation of patterns into specific projects, to instantiate design descriptions, and to generate code. However, they leave the selection of the most adequate pattern to model each variation point in the hands of the framework designer. Although this is obviously a creative task, if variation points are modeled during design tools that assist the systematization of the selection of the best modeling technique for each variation point may be constructed, simplifying the job of the framework designer.

5 Conclusions and Future Work

The standardization of the UML modeling language makes it attractive as a design notation for modeling OO frameworks. This paper shows that UML today lacks constructs to explicitly represent and classify framework variation points and their instantiation restrictions. The proposed extensions to the UML 1.3 design language address this problem representing variation points through appropriate markings. They make the framework design more explicit and therefore easier to understand and instantiate. The extensions have been defined by applying the UML extension mechanisms.

Although the extensions described in this paper have been used to model frameworks successfully [11], they are neither complete nor the only ones that may be applied to framework development. This paper discusses how to improve UML-F to provide additional extensions and a systematic approach to apply these extensions to different kinds of UML diagrams. Furthermore, it is of interest to understand that relationship of UML-F with similar kinds of variability problems, such as presented in [6].

The new UML-F elements are not concerned with how to implement the variability and extensibility aspects of the framework, but just with how to appropriately represent them at the design level. Furthermore, through use of this kind of extensions it is more likely that the framework user will not have to go into the detailed internals of a framework, being able to use it in a more black-box manner. Consequently, the diagrams give us a more abstract and concise representation of a framework, when compared to standard OOADM diagrams.

The most important claims of this paper is that frameworks should be modeled through appropriate design constructs that allow the representation of variation points and their intended behavior. The extended class diagrams and sequence diagrams facilitate the definition of adequate documentation, which may be used to assist the framework developer in modeling the variation points and the framework user in identifying these points during instantiation.

The extensions allow for the definition of supporting tools that may partially automate the development and instantiation activities. Appropriate tool assistance should also lead to a better time-to-market, reduced software costs, and higher software quality.

References

1. D. Bellin and S. Simone, *The CRC Card Book*, Addison Wesley Longman, 1997.
2. S. Berner, M. Glinz, S. Joos, “A Classification of Stereotypes for Object-Oriented Modeling Languages”, UML’99, *LNCS 1723*, Springer-Verlag, 249-264, 1999.
3. F. Budinsky, M. Finnie, J. Vlissides, and P. Yu, “Automatic Code Generation from Design Patterns”, *Object Technology*, 35(2), 1996.

4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
5. J. Coplien, “Broadening beyond objects to patterns and other paradigms”, *ACM Computing Surveys*, 28(4es), 152, 1996.
6. J. Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, 1999.
7. D. D’Souza, A. Sane, and A. Birchenough, “First-class Extensibility for UML – Packaging of Profiles, Stereotypes, Patterns”, UML’99, *LNCS 1723*, Springer-Verlag, 265-277, 1999.
8. D. D’Souza and A. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley, 1997.
9. A. Eden, J. Gil, and A. Yehudai, “Precise Specification and Automatic Application of Design Patterns”, ASE’97, *IEEE Press,* 1997.
10. G. Florijin, M. Meijers, P. van Winsen, “Tool Support for Object-Oriented Patterns”, ECOOP’97, *LNCS 1241,* Springer-Verlag, 472-495, 1997.
11. M. Fontoura, “A Systematic Approach for Framework Development“, Ph.D. Thesis, Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Brazil (PUC-Rio), 1999.
12. M. Fontoura, L. Moura, S. Crespo, and C. Lucena, “ALADIN: An Architecture for Learningware Applications Design and Instantiation”, Technical Report MCC34/98, Computer Science Department, Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Brazil (PUC-Rio), 1998.
13. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson, “Hooking into Object-Oriented Application Frameworks”, ICSE’97, *IEEE Press*, 491-501, 1997.
14. G. Froehlich, H. Hoover, L. Liu, and P. Sorenson, “Requirements for a Hooks Tool”, (http://www.cs.ualberta.ca/~softeng/papers/papers.htm).
15. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
16. D. Hamu and M. Fayad, ”Achieving Bottom-Line Improvements with Enterprise Frameworks“, *Communications of ACM*, 41(8), 110-113, 1998.
17. W. Harrison and H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”, OOPSLA’93, *ACM Press*, 411-428, 1993.
18. R. Helm, I. Holland, and D. Gangopadhyay, “Contracts: Specifying Behavioral Composition in Object-Oriented Systems”, OOPSLA/ECOOP’98, Norman Meyrowitz (ed.), *ACM Press*, 169-180, 1990.
19. I. Holland, “The Design and Representation of Object-Oriented Components”, Ph.D. Dissertation, Computer Science Department, Northeastern University, 1993.
20. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, ECOOP’96, *LNCS 1241*, 220-242, 1997.
21. G. Kiczales, J. des Rivieres, and D. Bobrow, *The Art of Meta-object Protocol*, MIT Press, 1991.
22. G. Krasner and S. Pope, “A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80”, *Journal of Object-Oriented Programming*, 1(3), 26-49, 1988.
23. T. Meijler, S. Demeyer, and R. Engel, “Making Design Patterns Explicit in FACE – A Framework Adaptative Composition Environment”, ESEC’97, *LNCS 1301,* Springer-Verlag, 94-111, 1997.
24. M. Mezini and K. Lieberherr, “Adaptative Plug-and-Play Components for Evolutionary Software Development”, OOPSLA’98, *ACM Press*, 97-116, 1998.
25. OMG, “OMG Unified Modeling Language Specification V.1.3”, 1999 (http://www.rational.com/uml).
26. D. Parnas, P. Clements, and D. Weiss, “The Modular Structure of Complex Systems”, *IEEE Transactions on Software Engineering,* SE-11, 259-266, 1985.

27. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1995.
28. W. Pree, *Framework Patterns*, Sigs Management Briefings, 1996.
29. T. Reenskaug, P. Wold, and O. Lehne, *Working with objects,* Manning, 1996.
30. D. Riehle and T. Gross, "Role Model Based Framework Design and Integration", OOPSLA'98, *ACM Press*, 117-133, 1998.
31. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Clifs, 1994.
32. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1998.
33. B. Rumpe, *A Note on Semantics*, Proceedings of Second ECOOP Workshop on Precise Behavioral Semantics, 1998.
34. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills, The Amsterdam Manifesto on OCL, Technical Report, Institute for Software Engineering, Technische Universität München, 1999.
35. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, A. Wills, Defining UML Family Members with Prefaces, TOOLS Pacific'99, *IEEE Press,* 1999.
36. J. Vlissides, *Pattern Hatching: Design Patterns Applied*, Software Patterns Series, Addison-Wesley, 1998.
37. R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.

Extending Moby with Inheritance-Based Subtyping

Kathleen Fisher[1] and John Reppy[2]

[1] AT&T Labs — Research
180 Park Avenue, Florham Park, NJ 07932 USA
kfisher@research.att.com
[2] Bell Laboratories, Lucent Technologies
700 Mountain Avenue, Murray Hill, NJ 07974 USA
jhr@research.bell-labs.com

Abstract. Classes play a dual role in mainstream statically-typed object-oriented languages, serving as both object generators and object types. In such languages, inheritance implies subtyping. In contrast, the theoretical language community has viewed this linkage as a mistake and has focused on subtyping relationships determined by the structure of object types, without regard to their underlying implementations. In this paper, we explore why inheritance-based subtyping relations are useful and we present an extension to the MOBY programming language that supports both inheritance-based and structural subtyping relations. In addition, we present a formal accounting of this extension.

1 Introduction

There is a great divide between the study of the foundations of object-oriented languages and the practice of mainstream object-oriented languages like JAVA[AG98] and C++[Str97]. One of the most striking examples of this divide is the rôle that class inheritance plays in defining subtyping relations. In most foundational descriptions of OO languages, and in the language designs that these studies have informed, inheritance does not define any subtyping relation, whereas in languages like JAVA and C++, inheritance defines a subtyping hierarchy. What is interesting about this distinction is that there are certain idioms, such as friend functions and binary methods, that are natural to express in an inheritance-based subtyping framework, but which require substantial complication to handle in a structural subtyping framework. It is important not to confuse inheritance-based subtyping with *by-name* subtyping, where the subtyping relationships between object types are explicitly declared (*e.g.*, JAVA's interfaces). While both inheritance-based and by-name subtyping avoid the *accidental subtyping* problem that structural subtyping suffers from,[1] the type names in an inheritance-based scheme are tied to specific implementations, whereas multiple, unrelated classes may be declared to implement the same type name in a by-name scheme.

In this paper, we explore why inheritance-based subtyping relations are useful, and we present an extension to the MOBY programming language [FR99a] that supports such subtyping relations by adding *class types*. While inheritance-based subtyping can

[1] The common example is a cowboy and widget that both have draw methods.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 83–107, 2000.

© Springer-Verlag Berlin Heidelberg 2000

be found in JAVA, C++, and other languages, the approach we take in Extended MOBY has several novel aspects. The most significant of these is that class types can be in an inheritance-based subtyping relationship even when their corresponding object types are not in the corresponding structural subtyping relationship. We also present a formal accounting of an object calculus that models the typing issues presented by Extended MOBY.

We begin by examining the common object-oriented idiom of *friend functions* and exploring how one might implement this idiom in MOBY [FR99a], which is a language with only structural subtyping. This example illustrates the deficiency of relying solely on structural subtyping in the language design. We then describe Extended MOBY and class types in Section 3. We show how this extension supports a number of common idioms, such as friend functions, binary methods, and object cloning. In Section 4, we present XMOC, an object calculus that models the type system of Extended MOBY. Specifically, XMOC supports both structural and inheritance-based subtyping, as well as privacy. To validate the design of Extended MOBY, we prove the type soundness of XMOC. In Section 5 we describe related work and we conclude in Section 6.

2 The Problem with Friends

Both C++ and JAVA have mechanisms that allow some classes and functions to have greater access privileges to a class's members than others. In C++, a class grants this access by declaring that certain other classes and functions are *friends*. In JAVA, members that are not annotated as `public`, `protected`, or `private` are visible to other classes in the same package, but not to those outside the package. In this section, we examine how to support this idiom in MOBY, which has only structural subtyping. This study demonstrates that while it is possible to encode the friends idiom in a language with only structural subtyping, the resulting encoding is not very appealing.

MOBY is a language that combines an ML-style module system with classes and objects [FR99a]. Object types are related by structural subtyping, which is extended to other types in the standard way. One of MOBY's most important features is that it provides flexible control over class-member visibility using a combination of the class and module mechanisms. The class mechanism provides a visibility distinction between clients that use the class to create objects and clients that use the class to derive subclasses, while the module mechanism provides hiding of class members via signature matching. The names of class members that are hidden by signature matching are truly private and can be redefined in a subclass. A brief introduction to MOBY is given in Appendix A.

2.1 Friends via Partial Type Abstraction

A standard way to program friends is to use partially abstract types [PT93, KLM94]. For example, Figure 1 gives the MOBY code for an implementation of a `Bag` class that has a `union` function as a friend. In this example, we have ascribed the `BagM` module with a signature that makes the `Rep` type partially abstract to the module's clients. Outside the module, if we have an object of type `Rep`, we can use both the `union` function and the `add` method (since `Rep` is a subtype of `Bag`), but we cannot access the `items` field.

```
module BagM : {
  type Rep <: Bag
  objtype Bag { meth add : Int -> Unit }
  val union : (Rep, Rep) -> Unit
  val mkBag : Unit -> Rep
} {
  class Bag {
    public field items : var List(Int)
    public meth add (x : Int) -> Unit {
      self.items := x :: self.items
    }
    public maker mk () { field items = Nil }
  }
  objtype Rep = typeof(Bag)
  objtype Bag { meth add : Int -> Unit }
  fun union (s1 : Rep, s2 : Rep) -> Unit {
    List.app s1.add s2.items
  }
  fun mkBag () -> Rep = new mk()
}
```

Fig. 1. Bags and friends using type abstraction

Inside the module, the `Rep` type allows access to all of the members of the `Bag` class;[2] the implementation of the `union` function exploits this access. Note that the `items` field is `public` inside the `BagM` module, but is not part of `BagM`'s interface outside the module. Thus, objects created from subclasses of `Bag` are not known to be structural subtypes of `Rep`.

Unfortunately, this approach only works for *final* classes. If we want to extend the `Bag` class, we must reveal the class in the signature of the `BagM` module (as is done in Figure 2). In this version, an object created using the `mk` maker cannot be used as an argument to the `union` function. This limitation also applies to objects created from subclasses of `Bag`.

2.2 Friends via Representation Methods

To support friends and class extension in the same class requires a public mechanism for mapping from an object to its abstract representation type. With this mechanism, we can recover the representation type required by the friend functions. For example, suppose we extend our `Bag` class to include a method that returns the number of items in the bag. We call this new class `CBag` (for counting bag), and we want to use the `union` function on objects created from the `CBag` class. Figure 3 presents this new implementation. Notice that we have added a public method `bagRep` to the interface of the `Bag` class, which returns `self` at the representation type (`Rep`). To apply the `union` function to two bags

[2] The Moby notation `typeof(`*C*`)` is shorthand for the object type that consists of the public members of class *C*.

```
module BagM : {
  type Rep <: typeof(Bag)
  class Bag {
    public meth add : Int -> Unit
    public maker mk of Unit
  }
  val union : (Rep, Rep) -> Unit
  val mkBag : Unit -> Rep
} {
  ...
}
```

Fig. 2. Revealing the Bag class

b1 and b2, we write “Bag.union (b1.bagRep(), b2.bagRep()).” This expression works even when b1 and/or b2 are counting bags.

Although this example does not include friends for the CBag class, we have included the representation method in its interface, which illustrates the main weakness of this approach. Namely, for each level in the class hierarchy, we must add representation types and methods. These methods pollute the method namespace and, in effect, partially encode the class hierarchy in the object types. Furthermore, it suffers from the source-code version of the *fragile base-class* problem: if we refactor the class hierarchy to add a new intermediate class, we have to add a new representation method, which changes the types of the objects created below that point in the hierarchy. While this encoding approach appears to be adequate for most of the examples that require a strong connection between the implementation and types, it is awkward and unpleasant.

3 Extended Moby

In the previous section, we showed how we can use abstract representation types and representation methods to tie object types to specific classes. From the programmer’s perspective, a more natural approach is to make the classes themselves serve the rôle of types when this connection is needed. In this section, we present an extension of Moby [FR99a] that supports such *class types* and *inheritance-based subtyping*. Intuitively, an object has a class type #C if the object was instantiated from C or one of its descendants. Inheritance-based subtyping is a form of by-name subtyping that follows the inheritance hierarchy. We illustrate this extension using several examples.

3.1 Adding Inheritance-Based Subtyping

Inheritance-based subtyping requires four additions to Moby’s type system, as well as a couple of changes to the existing rules:

For any class C, we define #C to be its *class type*, which can be used as a type in any context that is in C’s scope. Note that the meaning of a class type depends on

```
module BagM : {
  type Rep <: typeof(Bag)
  class Bag : {
    public meth add : Int -> Unit
    public meth bagRep : Unit -> Rep
    public maker mkBag of Unit
  }
  val union : (Rep, Rep) -> Unit
} {
  class Bag {
    public field items : var List(Int)
    public meth add (x : Int) -> Unit {
      self.items := x :: self.items
    }
    public meth bagRep () -> Rep { self }
    public maker mkBag () { field items = Nil }
  }
  objtype Rep = typeof(Bag)
  fun union (s1 : Rep, s2 : Rep) -> Unit {
    List.app s1.add s2.items
  }
}

module CBagM : {
  type Rep <: typeof(CBag)
  class CBag : {
    public meth add : Int -> Unit
    public meth bagRep : Unit -> BagM.Rep
    public meth size : Unit -> Int
    public meth cbagRep : Unit -> Rep
    public maker mkCBag of Unit
  }
} {
  class CBag {
    inherits BagM.Bag
    public field nItems : var Int
    public meth add (x : Int) -> Unit {
      self.nItems := self.nItems+1;
      super.add(x)
    }
    public meth size () -> Int { self.nItems }
    public meth cbagRep () -> Rep { self }
    public maker mkCBag () { super mkBag(); field nItems = 0 }
  }
  objtype Rep = typeof(CBag)
}
```

Fig. 3. Bags and friends using representation methods

```
class B {
  public meth m1 () -> Int { ... }
  public meth m2 ...
  ...
}
class C {
  inherits B : { public meth m2  ... }
  public meth m1 () -> Bool { ... }
  maker mkC of Unit { ... }
  ...
}
```

Fig. 4. Example of reusing a private method name

its context. Inside a method body, the class type of the host class allows access to all members, whereas outside the class, only the public members can be accessed.

We extend class interfaces to allow an optional `inherits` clause. If in a given context, a class `C` has an interface that includes an "`inherits B`" clause, then we view `#C` as a subtype of `#B`. Omitting the `inherits` clause from `C`'s interface causes the relationship between `B` and `C` to be hidden.

We say that `#C` is a subtype of `typeof(C)` (this relation corresponds to Fisher's observation that implementation types are subtypes of interface types [Fis96]).

The existing typing judgements for method and field selection require the argument to have an object type. We add new judgements for the case where the argument has a class type. We add new rules, instead of adding subsumption to the existing rules, to avoid a technical problem that is described in Section 3.2.

When typing the methods of a class `C`, we give `self` the type `#C` (likewise, if `B` is `C`'s superclass, then `super` has the type `#B`).

When typing a `new` expression, we assign the corresponding class type to the result.

3.2 Inheritance-Based Subtyping vs. Privacy

There is a potential problem in the Extended Moby type system involving the interaction between inheritance-based subtyping and Moby's support for privacy. Because Moby allows signature ascription to hide object members (*e.g.*, the `items` field in Figure 2), `#C` can be a subtype of `#B` even when `typeof(C)` is not a subtype of `typeof(B)`. The problem arises in the case where class `C` has defined a method that has the same name as one of `B`'s private methods. Consider the code fragment in Figure 4, for example.[3] Given these definitions, how do we typecheck the expression: "`(new mkC()).m1()`?" If we allow subsumption on the left-hand side of the method selection, then there are two incompatible ways to typecheck this expression. To avoid this ambiguity, we disallow subsumption on the left-hand side of member selection, and instead have two different rules for typing member selection: one for the case where the object's type is a class

[3] This example uses a *class interface* annotation on the class `B`; this syntactic form avoids the need to wrap `B` in a module and signature to hide the `m2` method.

type and one for the case where the object's type is an object type. An alternative design is to restrict the inheritance-based subtyping to hold between #C and #B only when typeof(C) is a subtype of typeof(B). The downside is that such a restriction reduces the power of the mechanism; for example, the mixin encoding described in Section 3.6 would not work.

3.3 Friends Revisited

We can now revisit our bag class example using the inheritance-based subtyping features of Extended Moby. In this new implementation (see Figure 5), we use the class type #Bag instead of the Rep type, which allows us to simplify the code by both eliminating the Rep type and the representation method. Note that the interface for the CBag class includes an inherits clause that specifies that it is a subclass of Bag. This relation allows the union function to be used on values that have the #CBag type.

3.4 Binary Methods

Binary methods are methods that take another object of the same class as an argument [BCC+96]. There are a number of different flavors of binary methods, depending on how objects from subclasses are treated. Using class types, we can implement binary methods that require access to the private fields of their argument objects. For example, Figure 6 shows how the union function in the previous example can be implemented as a binary method.

3.5 Object Cloning

Another case where inheritance-based subtyping is useful is in the typing of *copy constructors*, which can be used to implement a user-defined object cloning mechanism.[4] Figure 7 gives an example of cloning in Extended Moby. Class B has a private field (pvtX), which makes object types insufficient to type check C's use of the copyB maker function. The problem arises because the object type associated with self in typechecking C does not have a pvtX field (because that field is private to B), but the copyB maker function requires one. Thus, we need the inheritance-based subtyping relationship to allow the copyC maker to pass self, typed with #C, as a parameter to the copyB maker. Because we know that C inherits from B, this application typechecks. We also exploit this subtyping relation when we override the clone method.

3.6 Encoding Mixins

Moby does not support any form of multiple inheritance, but with the combination of parameterized modules and class types, it is possible to encode mixins [BC90, FKF98]. In this encoding, a mixin is implemented as a class parameterized over its base class using a parameterized module. The class interface of the base class contains only those components that are necessary for the mixin. After applying the mixin to a particular

[4] Note that in Moby, constructors are called *makers*.

```
module BagM : {
  class Bag : {
    public meth add : Int -> Unit
    public maker mkBag of Unit
  }
  val union : (#Bag, #Bag) -> Unit
} {
  class Bag {
    public field items : var List(Int)
    public meth add (x : Int) -> Unit {
      self.items := x :: self.items
    }
    public maker mkBag () { field items = Nil }
  }
  fun union (s1 : #Bag, s2 : #Bag) -> Unit {
    List.app s1.add s2.items
  }
}

module CBagM : {
  class CBag : {
    inherits BagM.Bag
    public meth size : Unit -> Int
    public maker mkCBag of Unit
  }
} {
  class CBag {
    inherits BagM.Bag
    public field nItems : var Int
    public meth add (x : Int) -> Unit {
      self.nItems := self.nItems+1;
      super.add(x)
    }
    public meth size () -> Int { self.nItems }
    public maker mkCBag () { super mkBag(); field nItems = 0 }
  }
}
```

Fig. 5. Bags with friends in Extended Moby

base class, we create a new class that inherits from the mixed base class and uses the class types to reconstitute the methods of the base class that were hidden as a result of the module application. Without class types, it would not be possible to make the original class's methods visible again. For example, Figure 8 gives the encoding of a mixin class that adds a `print` method to a class that has a `show` method. After applying `PrintMix` to class `A`, we define a class `PrA` that reconstitutes `A`'s `anotherMeth` method. Notice that we need to use an explicit type constraint to convert the type of `self` from `#PrA` to `#A`, since we do not have subsumption at method dispatch.

```
class Bag {
  field items : var List(Int)
  public meth add (x : Int) -> Unit {
    self.items := x :: self.items
  }
  public meth union (s : #Bag) -> Unit {
    List.app self.add s.items
  }
  public maker mkBag () { field items = Nil }
}
```

Fig. 6. Bags with a binary `union` method.

```
class B : {
  public meth getX : Unit -> Int
  public meth clone : Unit -> #B
  public maker mkB of Int
  maker copyB of #B
} {
  public meth getX () -> Int { self.pvtX }
  public meth clone () -> #B { new copyB(self) }
  public maker mkB (x : Int) { field pvtX = x }

  field pvtX : Int
  maker copyB (orig : #B) { field pvtX = orig.pvtX }
}

class C {
  inherits B
  public meth clone () -> #C { new copyC(self) }
  public maker mkC (y : Int) { super mkB(y) }
  maker copyC (orig : #C) { super copyB(orig) }
}
```

Fig. 7. Cloning with privacy in Extended Moby

While this encoding is cumbersome, it illustrates the power of class types. Also, it might serve as the definition of a derived form that directly supports mixins.

3.7 Efficiency of Method Dispatch

Although it is not our main motivation, it is worth noting that method dispatch and field selection from an object with a class type can be implemented easily as a constant time operation. When the dispatched method is final in the class type, the compiler can eliminate the dispatch altogether and call the method directly. In contrast, when an object has an object type, the compiler knows nothing about the layout of the object, making access more expensive.

```
signature HAS_SHOW {
  type InitB
  class B : {
    meth show : Unit -> String
    maker mk of InitB
  }
}
module PrintMix (M : HAS_SHOW)
{
  class Pr {
    inherits M.B
    public meth print () -> Unit {
      ConsoleIO.print(self.show())
    }
    maker mk (x : InitB) { super mk(x) }
  }
}

class A {
  public meth show () -> String { "Hi" }
  public meth anotherMeth () -> Unit { ... }
  maker mk () { }
}

module P = PrintMix({type InitB = Unit; class B = A})

class PrA {
  inherits P.Pr
  public meth anotherMeth () -> Unit {
    (self : #A).anotherMeth()
  }
}
```

Fig. 8. Encoding mixins in Extended Moby

4 XMOC

We have developed a functional object calculus, called XMOC, that models the type system of Extended Moby and validates its design. XMOC supports both traditional structural subtyping and inheritance-based subtyping. In this section, we discuss the intuitions behind XMOC and state type soundness results; space considerations preclude a more detailed presentation. The full system is given in Appendices B and C.

4.1 Syntax

The term syntax of XMOC is given in Figure 9. An XMOC program consists of a sequence of class declarations terminated by an expression. Each declaration binds a

$$
\begin{array}{rcl}
p & ::= & d;\ p \\
  & \mid & e \\
d & ::= & \texttt{class}\ C\ (x : \tau)\ \{\!\!|\, b\ {m = \mu_m}^{m \in \mathcal{M}}\, |\!\!\} \\
  & \mid & \texttt{class}\ C : \sigma = C' \\
b & ::= & \\
  & \mid & \texttt{inherits}\ C(e); \\
\sigma & ::= & (\tau)\ \{\!\!|\, \texttt{inherits}\ B;\ {m : \tau_m}^{m \in \mathcal{M}}\, |\!\!\} \\
\tau & ::= & \alpha \\
  & \mid & \tau \rightarrow \tau' \\
  & \mid & \texttt{obj}\ \alpha\,.\{\!\!|\, {m : \tau_m}^{m \in \mathcal{M}}\, |\!\!\} \\
  & \mid & \texttt{\#}C
\end{array}
$$

$$
\begin{array}{rcl}
\mu & ::= & (x\ :\ \tau) \Rightarrow e \\
e & ::= & x \\
  & \mid & \texttt{fn}(x\ :\ \tau) \Rightarrow e \\
  & \mid & e(e') \\
  & \mid & \texttt{new}\ C(e) \\
  & \mid & \texttt{self} \\
  & \mid & e.m \\
  & \mid & \texttt{self!state} \\
  & \mid & e\,@\,C
\end{array}
$$

Fig. 9. Syntax of XMOC terms

class name to a class definition; we do not allow rebinding of class names. The set of class names includes the distinguished name **None**, which is used to denote the super-class of base classes. We follow the convention of using C for class names other than **None** and B for class names including **None**. Class declarations come in two forms. In the first, a class C is defined by extending an optional parent class (b) by overriding and adding methods. When no parent is specified, we say that C is a *base-class*; otherwise, we say that C *inherits* from its parent class. To model the notion of object state, we parameterize this form of class declaration by a variable x. When an object is created, the argument bound to x is bound into the representation of the object as its state. In addition, x is used to compute the argument for the superclass. Since an object's implementation is spread over a hierarchy of classes, it has a piece of state for each class in its implementation. Methods of a given class may access the state for that class using the form `self!state`. In the second form of class declaration, a class C can be derived from an existing class C' by *class-interface ascription*, which produces a class that inherits its implementation from C', but has a more restrictive class interface σ. A class interface σ specifies the type of the class's state variable, the name of the nearest revealed ancestor class (or **None**), and a typed list of available methods. The types of XMOC include type variables, function types, recursive object types, and class types.

Each method (μ) takes a single parameter and has an expression for its body. The syntax of expressions (e) includes variables, functions, function application, new object creation, the special variable `self` (only allowed inside method bodies), method dispatch, and access to the object's state. The last expression form ($e\,@\,C$) is an *object-view coercion*. Unlike Extended Moby, XMOC does not map the inheritance relation directly to the subtyping relation; instead we rely on object-view coercions to explicitly coerce the type of an expression from a class to one of its superclasses. This approach avoids the problem discussed in Section 3.2 without requiring two typing judgements for method dispatch. It is possible to automatically insert these coercions into the XMOC representation of a program as part of typechecking (such a translation is similar to the type-directed representation wrapping that has been done for polymorphic languages [Ler92]).

4.2 Dynamic Semantics

Evaluation of an XMOC program occurs in two phases. The first phase is defined by the *class linking* relation, written $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$, which takes a *dynamic class environment* $\mathcal{K}$ and links the left-most class definition in p to produce $\mathcal{K}'$. Class linking terminates with a residual expression once all of the class declarations have been linked. The second phase evaluates the residual expression to a value (assuming termination). This phase is defined by the expression evaluation relation, which we write as $\mathcal{K} \vdash e \hookrightarrow e'$. Defining the semantics of linking and evaluation requires extending the term syntax with run-time forms.

Correctly handling class-interface ascription provides the greatest challenge in defining the semantics for XMOC. Using this mechanism, a public method m in B can be made private in a subclass C, and subsequently m can be reused to name an unrelated method in some descendant class of C (recall the example in Figure 4). Methods inherited from B must invoke the original m method when they send the m message to `self`, while methods defined in D must get the new version. We use a variation of the Riecke-Stone dictionary technique [RS98, FR99b] to solve this problem. Intuitively, dictionaries provide the α-conversion needed to avoid capture by mapping method names to *slots* in method tables. To adapt this technique to XMOC, when we link a class C, we replace each occurrence of `self` in the methods that C defines with the object view expression "$\mathit{self} \,@\, C$." Rule 5 in Appendix B describes this annotation formally. At runtime, we represent each object as a pair of a raw object (denoted by meta-variable obj) and a view (denoted by a class name). The raw object consists of the object's state and the name of the defining class; this class name is used to find the object's method suite. The view represents the visibility context in which the message send occurs; those methods in scope in class C are available. To lookup method m in runtime object $\langle obj,\ C\rangle$, we use the dictionary associated with C in the class environment $\mathcal{K}$ to find the relevant slot. We use this slot to index into the method table associated with obj. Rule 3 formally specifies method lookup.

4.3 Static semantics

The XMOC typing judgements are written with respect to a static environment Γ, which consists of a set of bound type variables ($\mathcal{A}$), a subtype assumption map ($\mathcal{S}$), a class environment ($\mathcal{C}$), and a variable environment ($\mathcal{V}$). We define these environments and give the complete set of XMOC typing judgements in Appendix C. Here we briefly discuss some of the more important rules.

As mentioned earlier, each XMOC class name doubles as an object type. We associate such a type with an object whenever we instantiate an object from a class, according to the typing rule

$$\frac{(\mathcal{C}\ \mathrm{of}\ \Gamma)(C) = (\tau)\ \{\!|\ \mathtt{inherits}\ B;\ m : {\tau_m}^{m \in \mathcal{M}}\ |\!\}\qquad \Gamma \vdash e \rhd \tau' \qquad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash \mathtt{new}\ C(e) \rhd \mathtt{\#}C}$$

which looks up class C in Γ, infers a type τ' for the constructor argument e, and insures that this type is a subtype of the type of the class parameter τ.

In contexts that allow subsumption, we can treat a class type as an object type according to the following subtyping judgement:

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!\!|\, \texttt{inherits}\, B;\ m : \tau_m{}^{m\in\mathcal{M}} \,|\!\!\}}{\Gamma \vdash \texttt{\#}C <: \texttt{obj}\,\alpha\,\texttt{.}\,\{\!\!|\, m : \tau_m{}^{m\in\mathcal{M}} \,|\!\!\}}$$

This rule corresponds to the property that `#C` is a subtype of `typeof(C)` in Extended Moby. Note that unlike Extended Moby, we do not treat a class type $\texttt{\#}C$ as being a subtype of its superclass type. Instead we use an object view constraint, which is typed as follows:

$$\frac{\Gamma \vdash e \rhd \texttt{\#}C' \quad \mathcal{H}(\Gamma) \vdash C' \lessdot C}{\Gamma \vdash e\,@\,C \rhd \texttt{\#}C}$$

The judgement $\mathcal{H}(\Gamma) \vdash C' \lessdot C$ states that C' inherits from C in the class hierarchy $\mathcal{H}(\Gamma)$ (the meta-function $\mathcal{H}$ projects the static class hierarchy from the environment Γ). Because we do not treat inheritance directly as subtyping in XMOC, we only need one rule for typing method dispatch.

$$\frac{\Gamma \vdash e \rhd \tau \quad \Gamma \vdash \tau <: \texttt{obj}\,\alpha\,\texttt{.}\,\{\!\!|\, m : \tau_m{}^{m\in\mathcal{M}} \,|\!\!\} \quad m \in \mathcal{M}}{\Gamma \vdash e\,\texttt{.}\,m \rhd \tau_m[\alpha \mapsto \texttt{obj}\,\alpha\,\texttt{.}\,\{\!\!|\, m : \tau_m{}^{m\in\mathcal{M}} \,|\!\!\}]}$$

4.4 Soundness

We have proven the type soundness of XMOC and we outline this result here. We start with a *subject reduction* theorem, which states that the evaluation relation preserves the type of programs and class environments.

Theorem 1 *If $\Gamma \vdash \mathcal{K}$, and for a program p we have $\Gamma, \mathcal{H}(\mathcal{K}) \vdash p \rhd \Gamma_f, \tau_f$ and $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$, then there exists an environment Γ' and a type τ', such that*

$\Gamma', \mathcal{H}(\mathcal{K}') \vdash p' \rhd \Gamma_f, \tau'$,
$\Gamma' \vdash \tau' <: \tau_f$, *and*
$\Gamma' \vdash \mathcal{K}'$

Furthermore, if p is an expression, then $\Gamma_f = \Gamma'$.

The proof has two steps: we show that linking preserves types and produces a well-typed class environment, and then we show that expression evaluation preserves the type of expressions.

The next step is a *progress* theorem, which states that well-typed programs do not get *stuck*.

Theorem 2 *If $\Gamma \vdash \mathcal{K}$, and for a program p we have $\Gamma, \mathcal{H}(\mathcal{K}) \vdash p \rhd \Gamma_f, \tau_f$ with p and τ_f both closed, then either p is a value or there exists $\mathcal{K}'$ and p' such that $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$.*

Here we use $\mathcal{H}$ to project the *dynamic* class hierarchy from the dynamic class environment $\mathcal{K}$.

Definition 1 *We say that a program p* yields *$\mathcal{K}', p'$ if $\emptyset, p \rightarrowtail \mathcal{K}', p'$ and there does* not *exist $\mathcal{K}'', p''$ such that $\mathcal{K}', p' \rightarrowtail \mathcal{K}'', p''$.*

Finally, we show the type soundness of XMOC.

Theorem 3 *If $\emptyset \vdash p \triangleright \Gamma_f, \tau_f$ and p yields $\mathcal{K}', p'$, then p' is a value w, such that $\Gamma_f, \mathcal{H}(\mathcal{K}') \vdash w \triangleright \tau'$ with $\Gamma_f \vdash \tau' <: \tau_f$.*

5 Related Work

Our class types are motivated by the rôle that classes play in languages like C++ and JAVA. The main difference between Extended MOBY and the class types provided by these other languages is in the way that abstraction is supported. Extended MOBY allows partial hiding of inherited components using signature ascription, which means that `typeof(C)` may not be a subtype of `typeof(B)` even when `C` is known to inherit from `B` (see Section 3.2). JAVA does not support private inheritance, but C++ allows a class to *privately* inherit from another class, but in that case all of the base-class members are hidden and there is no subtyping relationship. Extended MOBY is more flexible, since it allows hiding to be done on a per-member basis. It also allows the class hierarchy to be hidden by omitting the `inherits` clause in class interfaces. In C++ and JAVA the full class hierarchy is manifest in the class types (except for C++'s private inheritance). Another point of difference is that Extended MOBY supports structural subtyping on object types; JAVA has object types (called interfaces), but subtyping is *by-name*. C++ does not have an independent notion of object type.

Fisher's Ph.D. dissertation [Fis96] is the earliest formalization of class types that we are aware of. In her work, each implementation is tagged with a row variable using a form of bounded existential rows. In our work, we adopt classes as a primitive notion and use the names of such classes in a fashion analogous to Fisher's row variables. A weakness of the earlier work is its treatment of private names; it provides no way to hide a method and then later add an unrelated method with the same name.

Our use of dictionaries to specify the semantics of method dispatch in the presence of privacy is adapted from the work of Riecke and Stone [RS98]. The main difference is that XMOC has an explicit notion of class and we introduce dictionaries as a side-effect of linking classes.

More recently, Igarashi *et al.* have described *Featherweight Java*, which is an object calculus designed to model the core features of JAVA's type system [IPW99]. Like our calculus, Featherweight Java has a notion of subtyping based on class inheritance. Our calculus is richer, however, in a number of ways. Our calculus models private members and narrowing of class interfaces. We also have a notion of structural subtyping and we relate the implementation and structural subtyping notions.

The notion of type identity based on implementation was present in the original definition of *Standard ML* in the form of *structure sharing* [MTH90]. The benefits of structure sharing were fairly limited and it was dropped in the 1997 revision of SML [MTHM97].

6 Conclusion

This paper presents an extension to MOBY that supports classes as types. We have illustrated the utility of this extension with a number of examples. We believe that Extended MOBY is the first design that incorporates types for objects that range from class types to structural object types.[5] While this flexibility goes beyond what is found in other languages, the most interesting aspect of the design is the interaction between privacy (specified via module signature matching) and inheritance-based subtyping. Because we allow the inherits relation between two class types to be visible even when their corresponding object types are not in a subtyping relationship, one can use class types to recover access to hidden methods. This property enables more flexible use of parameterized modules in combining class implementations, as illustrated in Section 3.6.

We have also developed a formal model of this extension and have proven type soundness for it. We are continuing to work on improving our formal treatment of class types and implementation-based inheritance. One minor issue is that XMOC requires that class names be unique in a program; this restriction can be avoided by introducing some mechanism, such as *stamps*, to distinguish top-level names (*e.g.*, see Leroy's approach to module system semantics [Ler96]). We would also like to generalize the rule that relates class types with object types (rule 36 in Appendix C) to allow positive occurrences of $\#C$ to be replaced by the object type's bound type variable. While we believe that this generalization is sound, we have not yet proven it.

References

[AG98] Arnold, K. and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1998.

[BC90] Bracha, G. and W. Cook. Mixin-based inheritance. In *ECOOP'90*, October 1990, pp. 303–311.

[BCC$^+$96] Bruce, K., L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *TAPOS*, **1**(3), 1996, pp. 221–242.

[Fis96] Fisher, K. *Type Systems for Object-oriented Programming Languages*. Ph.D. dissertation, Stanford University, August 1996.

[FKF98] Flatt, M., S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL'98*, January 1998, pp. 171–183.

[FR99a] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *PLDI'99*, May 1999, pp. 37–49.

[FR99b] Fisher, K. and J. Reppy. Foundations for MOBY classes. *Technical Memorandum*, Bell Labs, Lucent Technologies, Murray Hill, NJ, February 1999.

[IPW99] Igarashi, A., B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA'99*, November 1999, pp. 132–146.

[KLM94] Katiyar, D., D. Luckham, and J. Mitchell. A type system for prototyping languages. In *POPL'94*, January 1994, pp. 138–161.

[Ler92] Leroy, X. Unboxed objects and polymorphic typing. In *POPL'92*, January 1992, pp. 177–188.

[5] JAVA is close to Extended Moby in this respect, but interface subtyping relations in JAVA must be declared ahead of time by the programmer, whereas object-type subtyping in MOBY is based on the structure of the types.

[Ler96] Leroy, X. A syntactic theory of type generativity and sharing. *JFP*, **6**(5), September 1996, pp. 1–32.

[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML — Revised 1997*. The MIT Press, Cambridge, MA, 1997.

[PT93] Pierce, B. C. and D. N. Turner. Statically typed friendly functions via partially abstract types. *Technical Report ECS-LFCS-93-256*, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[RS98] Riecke, J. G. and C. Stone. Privacy via subsumption. In *FOOL5*, January 1998. A longer version will appear in TAPOS.

[Str97] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

A A Brief Introduction to MOBY

This appendix provides a brief introduction to some of MOBY's features to help the reader understand the examples in the paper. A more detailed description of MOBY object-oriented features can be found in [FR99a].

MOBY programs are organized into a collection of *modules*, which have *signatures*. A module's signature controls the visibility of its components. Signatures are the primary mechanism for data and type abstraction in MOBY. To support object-oriented programming, MOBY provides *classes* and *object types*. The following example illustrates these features:

```
module M : {
  class Hi : {
    public meth hello : Unit -> Unit
    public maker mk of String
  }
  val hi : typeof(Hi)
} {
  fun pr (s : String) -> Unit { ConsoleIO.print s }
  class Hi {
    field msg : String
    public meth hello () -> Unit {
      pr "hello "; pr self.msg; pr "\n"
    }
    public maker mk (s : String) { field msg = s }
  }
  val hi : typeof(Hi) = new mk "world"
}
```

This code defines a module `M` that is constrained by a signature with two specifications: the class `Hi` and the value `hi`. The interface of the `Hi` class specifies that it has two public components: a method `hello` and a maker `mk` ("maker" is the MOBY name for constructor functions). The signature specifies that `hi` is an object; the type expression "`typeof(Hi)`" denotes the object type induced by reading off the public methods and fields of the class `Hi`. It is equivalent to the object type definition

```
objtype HiTy { public meth hello : Unit -> Unit }
```

The body of `M` defines a function `pr` for printing strings, and the definitions of `Hi` and `hi`. Since `pr` is not mentioned in the signature of `M`, it is not exported. Note that the `Hi` class has a field `msg` in its definition. Since this field does not have a `public` annotation, it is only visible to subclasses. Furthermore, since `msg` is not mentioned in `M`'s signature, it is not visible to subclasses of `Hi` outside of `M`. Thus, the `msg` field is *protected* inside `M` and is *private* outside. This example illustrates MOBY's use of module signatures to implement private class members.

B Dynamic Semantics of XMOC

B.1 Notation

If A and B are two sets, we write $A \setminus B$ for the set difference and $A \pitchfork B$ if they are *disjoint*. We use the notation $A \overset{\text{fin}}{\to} B$ to denote the set of finite maps from A to B. We write $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ for the finite map that maps a_1 to b_1, etc. For a finite map f, we write $\text{dom}(f)$ for its domain and $\text{rng}(f)$ for its range. If f and g are finite maps, we write $f \pm g$ for the finite map

$$\{x \mapsto g(x) \mid x \in \text{dom}(g)\} \cup \{x \mapsto f(x) \mid x \in \text{dom}(f) \setminus \text{dom}(g)\}$$

We write $t[x \mapsto t']$ for the *capture-free* substitution of t' for x in the term t.

B.2 Syntax

We use the following classes of identifiers in the syntax of XMOC.

$\alpha \in \textsc{TyVar}$	type variables
$\tau \in \textsc{Type}$	types
$\sigma \in \textsc{Interface}$	class interfaces
$B, C \in \textsc{ClassName} = \{\mathbf{None}, \ldots\}$	class names
$\#B, \#C \in \textsc{ClassType}$	class types
$m \in \textsc{MethName}$	method names
$\mathcal{M} \in \text{Fin}(\textsc{MethName})$	method name sets
$x \in \textsc{Var}$	variables
$\mu \in \textsc{MethBody}$	method bodies
$e \in \textsc{Exp}$	expressions
$obj \in \textsc{ObjExp}$	object expressions

We follow the convention of using C when referring to a class name other than **None**. Figure 10 describes the full syntax of XMOC. In this grammar, we mark the run-time forms with a $(*)$ on the right. The obj form represents a raw object at run-time: C is its instantiating class, e denotes its local state, and obj represents the portion of the object inherited from its parent class. The methods associated with class C are found in the dynamic class environment, defined below.

$$\begin{array}{rcl}
p & ::= & d;\ p \\
 & \mid & e \\
d & ::= & \texttt{class}\ C\ (x : \tau)\ \{\!|\ b\ \ m = {\mu_m}^{m \in \mathcal{M}}\ |\!\} \\
 & \mid & \texttt{class}\ C : \sigma = C' \\
\sigma & ::= & (\tau)\ \{\!|\ \texttt{inherits}\ B;\ \ m : {\tau_m}^{m \in \mathcal{M}}\ |\!\} \\
\tau & ::= & \alpha \\
 & \mid & \tau \rightarrow \tau' \\
 & \mid & \texttt{obj}\ \alpha\,\texttt{.}\,\{\!|\ m : {\tau_m}^{m \in \mathcal{M}}\ |\!\} \\
 & \mid & \texttt{\#}B \\
b & ::= & \\
 & \mid & \texttt{inherits}\ C(e);
\end{array}$$

$$\begin{array}{rcll}
\mu & ::= & (x\ :\ \tau) \Rightarrow e & \\
e & ::= & x & \\
 & \mid & \texttt{fn}(x\ :\ \tau) \Rightarrow e & \\
 & \mid & e(e') & \\
 & \mid & \texttt{new}\ C(e) & \\
 & \mid & \texttt{self} & \\
 & \mid & e\texttt{!state} & \\
 & \mid & e\ @\ C & \\
 & \mid & e\,\texttt{.}\,m & \\
 & \mid & \langle obj,\ C\rangle & (*) \\
obj & ::= & \mathbf{None} & (*) \\
 & & C :: \{\!|\ e;\ obj\ |\!\} & (*) \\
 & & C(e) & (*)
\end{array}$$

Fig. 10. The full syntax of XMOC

B.3 Evaluation

To define the evaluation relation, we need some additional definitions:

$$\begin{array}{rll}
 & \textsc{Slot} & \text{method suite slots} \\
\phi \in & \textsc{Dict} = \textsc{MethName} \overset{\text{fin}}{\rightarrow} \textsc{Slot} & \text{dictionaries} \\
\mu T \in & \textsc{MethSuite} = \textsc{Slot} \overset{\text{fin}}{\rightarrow} \textsc{MethBody} & \text{method suites} \\
H \in & \textsc{Hierarchy} = \textsc{ClassName} \overset{\text{fin}}{\rightarrow} \textsc{ClassName} & \text{class hierarchy}
\end{array}$$

A *dictionary* ϕ is a one-to-one finite function from method names to slots. We say that ϕ is a *slot assignment* for a set of method names M if and only if $\text{dom}(\phi) = M$. A *method suite* is a finite function from slots to method bodies. A class hierarchy describes the inheritance relationship between classes by mapping each class name to the name of its superclass.

The dynamic semantics is split into two phases; the first phase *links* the class declarations to produce a dynamic class environment and a residual expression. The dynamic class environment is a finite map from class names to a tuple, storing the name of the parent class, a constructor expression for initializing the inherited state of instantiated objects, a method suite, which maps slot numbers to method bodies, and a dictionary, which maps method names to slot numbers.

$$\mathcal{K} \in \textsc{DynClassEnv} = \textsc{ClassName} \overset{\text{fin}}{\rightarrow} (\textsc{ClassName} \times \textsc{Exp} \times \textsc{MethSuite} \times \textsc{Dict})$$

We write the linking relation as $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$. The second phase *evaluates* the residual expression using the dynamic class environment to instantiate objects and resolve method dispatch. The evaluation relation is written formally as $\mathcal{K}, e \rightarrowtail \mathcal{K}, e'$, but we abbreviate this notation to $\mathcal{K} \vdash e \hookrightarrow e'$ when convenient.

B.4 Dynamic Class Hierarchy

We construct the dynamic class hierarchy from the dynamic class environment $\mathcal{K}$ as follows:

$$\mathcal{H}(\mathcal{K}) = \{C \mapsto B \mid C \in \mathrm{dom}(\mathcal{K}) \text{ and } \mathcal{K}(C) = (B,\ \_,\ \_,\ \_)\}$$

B.5 Evaluation Contexts and Values

The dynamic semantics is specified using the standard technique of evaluation contexts. We distinguish two kinds of contexts in the specification of the evaluation relation: *expression contexts*, E, and *object initialization contexts*, F. The syntax of these contexts is as follows:

$$\begin{aligned} E &::= [\,] \mid E(e) \mid w(E) \mid \mathtt{new}\ C(E) \mid E\mathtt{!state} \mid E\,@\,C \mid E\,.m \mid \langle F,\ C\rangle \\ F &::= [\,] \mid C(E) \mid C :: \{\!|\, E;\ \mathit{obj}\,|\!\} \mid C :: \{\!|\, w;\ F\,|\!\} \end{aligned}$$

We also define the syntactic class of *values* (w) and *object values* (ov) by the following grammar:

$$\begin{aligned} w &::= \mathtt{fn}(x : \tau) \Rightarrow e \mid \langle ov,\ C\rangle \\ ov &::= \mathbf{None} \mid C :: \{\!|\, w;\ ov\,|\!\} \end{aligned}$$

B.6 Field Lookup

The auxiliary *field lookup* relation $C_v \vdash C :: \{\!|\, w;\ ov\,|\!\} \leadsto_f w'$ is used to specify the semantics of state selection.

$$\frac{}{C \vdash C :: \{\!|\, w;\ ov\,|\!\} \leadsto_f w} \quad (1) \qquad \frac{C_v \neq C \qquad C_v \vdash ov \leadsto_f w'}{C_v \vdash C :: \{\!|\, w;\ ov\,|\!\} \leadsto_f w'} \quad (2)$$

B.7 Method Lookup

We define the auxiliary *method lookup* relation $\mathcal{K} \vdash \langle ov,\ C_v\rangle\,.m \leadsto_m w$, which specifies how method dispatching is resolved, as follows:

$$\frac{\mathcal{K}(C_v) = (\_,\ \_,\ \_,\ \phi) \qquad \mathcal{K}(C) = (\_,\ \_,\ \mu T,\ \_) \qquad \mu T(\phi(m)) = (x : \tau) \Rightarrow e}{\mathcal{K} \vdash \langle C :: \{\!|\, w;\ ov\,|\!\},\ C_v\rangle\,.m \leadsto_m \mathtt{fn}(x : \tau) \Rightarrow e[\mathit{self} \mapsto C :: \{\!|\, w;\ ov\,|\!\}]} \quad (3)$$

B.8 Class Linking

The evaluation relation for class linking is defined by the following three rules. The first describes the case of a base-class declaration. Note that since a base class has no superclass, it does not require a parent initialization expression.

$$\begin{aligned}
&\mathcal{K}, \texttt{class}\ C\ (x:\tau)\ \{\!|\ m = \mu_m{}^{m\in\mathcal{M}}\ |\!\};\ p \ \rightarrowtail\ \mathcal{K}\pm\{C \mapsto (\mathbf{None}, \text{–}, \mu T, \phi)\}, p\\
&\text{where}\ \overline{\mu}_m = \mu_m[\mathit{self} \mapsto \mathit{self} \,@\, C]\\
&\qquad \phi \text{ is a slot assignment for } M.\\
&\qquad \mu T = \{\phi(m) \mapsto \overline{\mu}_m \mid m \in M\}
\end{aligned} \tag{4}$$

The second rule handles the case of a subclass declaration.

$$\begin{aligned}
&\mathcal{K}, \texttt{class}\ C\ (x:\tau)\ \{\!|\ \texttt{inherits}\ C'(e);\ m = \mu_m{}^{m\in\mathcal{M}}\ |\!\};\ p\\
&\quad \rightarrowtail \quad \mathcal{K}\pm\{C \mapsto (C', \texttt{fn}(x:\tau) \Rightarrow e, \mu T_C, \phi_C)\}, p\\
&\text{where}\ \mathcal{K}(C') = (\_, \_, \mu T_{C'}, \phi_{C'})\\
&\qquad \phi \text{ is a slot assignment for } M \setminus \mathrm{dom}(\phi_{C'}) \text{ such that } \mathrm{rng}(\phi) \pitchfork \mathrm{dom}(\mu T_{C'})\\
&\qquad \phi_C = \phi_{C'} \cup \phi\\
&\qquad \overline{\mu}_m = \mu_m[\mathit{self} \mapsto \mathit{self} \,@\, C]\\
&\qquad \mu T_C = \mu T_{C'} \pm \{\phi_C(m) \mapsto \overline{\mu}_m \mid m \in M\}
\end{aligned} \tag{5}$$

The third linking rule describes class signature ascription.

$$\begin{aligned}
&\mathcal{K}, \texttt{class}\ C : (\tau)\ \{\!|\ \texttt{inherits}\ B;\ m : \tau_m{}^{m\in\mathcal{M}}\ |\!\} = C';\ p\\
&\quad \rightarrowtail \quad \mathcal{K}\pm\{C \mapsto (C', \texttt{fn}(x:\tau) \Rightarrow x, \mu T_C, \phi_C)\}, p\\
&\text{where}\ \mathcal{K}(C') = (\_, \_, \mu T_{C'}, \phi_{C'})\\
&\qquad \mu T_C = \mu T_{C'}\\
&\qquad \phi_C = \phi_{C'} | M
\end{aligned} \tag{6}$$

B.9 Expression Evaluation

The following rules specify the semantics of expression evaluation:

$$\mathcal{K} \vdash E[\texttt{fn}(x:\tau) \Rightarrow e(w)] \quad \hookrightarrow \quad E[e[x \mapsto w]] \tag{7}$$

$$\mathcal{K} \vdash E[\texttt{new}\ C(w)] \quad \hookrightarrow \quad E[\langle C(w), C\rangle] \tag{8}$$

$$\mathcal{K} \vdash E[\langle ov, C_v\rangle \texttt{!state}] \quad \hookrightarrow \quad E[w] \qquad \text{where } C_v \vdash ov \leadsto_f w \tag{9}$$

$$\mathcal{K} \vdash E[\langle ov, C_v\rangle \,@\, C'_v] \quad \hookrightarrow \quad E[\langle ov, C'_v\rangle] \tag{10}$$

$$\mathcal{K} \vdash E[w\,.\,m] \quad \hookrightarrow \quad E[w'] \qquad \text{where } \mathcal{K} \vdash w\,.\,m \leadsto_m w' \tag{11}$$

$$\begin{aligned}
&\mathcal{K} \vdash F[C(w)] \quad \hookrightarrow \quad F[C :: \{\!|\ w;\ \mathit{obj}\ |\!\}]\\
&\text{where} \quad \mathit{obj} = \begin{cases} \mathbf{None} & \text{if } \mathcal{K}(C) = (\mathbf{None}, \_, \_, \_)\\ C'(e(w)) & \text{if } \mathcal{K}(C) = (C', e, \_, \_) \end{cases}
\end{aligned} \tag{12}$$

C Typing Rules for XMOC

The typing rules for XMOC are written with respect to an environment Γ, which has four parts:

$\mathcal{A} \in \textsc{TyVarSet} = \mathrm{Fin}(\textsc{TyVar})$	bound type variables
$\mathcal{S} \in \textsc{SubtyEnv} = \textsc{TyVar} \overset{\text{fin}}{\to} \textsc{Type}$	subtyping assumptions
$\mathcal{C} \in \textsc{ClassEnv} = \textsc{ClassName} \overset{\text{fin}}{\to} \textsc{Interface}$	class typing environment
$\mathcal{V} \in \textsc{VarEnv} = \textsc{Var} \overset{\text{fin}}{\to} \textsc{Type}$	variable typing environment
$\Gamma \in \textsc{Env} =$ $\textsc{TyVarSet} \times \textsc{SubtyEnv} \times \textsc{ClassEnv} \times \textsc{VarEnv}$	typing environment

C.1 Static Class Hierarchy

We construct the static class hierarchy from the environment Γ as follows:

$$\mathcal{H}(\Gamma) = \{(C \mapsto B) \mid (\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!|\, \texttt{inherits}\,B;\ m : {\tau_m}^{m \in \mathcal{M}} \,|\!\}\}$$

C.2 Judgement Forms

$\Gamma \vdash \tau \rhd \mathbf{Ok}$	Type τ is *well-formed* w.r.t. Γ.	(rules 13–17)
$\Gamma \vdash \sigma \rhd \mathbf{Ok}$	Class interface σ is *well-formed* w.r.t. Γ.	(rule 18)
$H \vdash \mathbf{Ok}$	Class hierarchy H is *well-formed.*	(rule 19)
$\Gamma \vdash \mathbf{Ok}$	Environment Γ is *well-formed.*	(rule 20)
$\Gamma \vdash \tau = \tau'$	Type τ is *equal* to τ'.	(rules 21–26)
$H \vdash B_1 \lessdot B_2$	Class B_1 *inherits* from B_2.	(rules 27–30)
$\Gamma \vdash \tau <: \tau'$	Type τ is a *subtype* of τ'.	(rules 31–36)
$\Gamma \vdash \sigma <: \sigma'$	Class interface σ is a *subtype* of σ'.	(rule 37)
$\Gamma \vdash p \rhd \Gamma', \tau$	Program p *defines* environment Γ' and *has type* τ.	(rules 38–39)
$\Gamma \vdash d \rhd \Gamma'$	Declaration d *defines* environment Γ'.	(rules 40–42)
$\Gamma \vdash \mu \rhd \tau$	Method μ *has type* τ.	(rule 43)
$\Gamma \vdash e \rhd \tau$	Expression e *has type* τ.	(rules 44–51)
$\Gamma, H \vdash obj \rhd \#B$	Parent class obj *has class type* $\#B$.	(rules 52–54)
$\Gamma, H \vdash e \rhd \tau$	Run-time expression e *has type* τ.	(rules 53 and 56)
$\Gamma, H, C \vdash con \rhd \sigma$	Constructor con, associated with class C, *has interface* σ.	(rules 57–58)
$\Gamma \vdash \mathcal{K}$	Static environment Γ *types* dynamic class environment $\mathcal{K}$.	(rule 59)

C.3 Well-Formedness Judgements

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \alpha \in (\mathcal{A}\,\mathrm{of}\,\Gamma)}{\Gamma \vdash \alpha \rhd \mathbf{Ok}} \tag{13}$$

$$\frac{\Gamma \vdash \mathbf{Ok}}{\Gamma \vdash \#\mathbf{None} \rhd \mathbf{Ok}} \tag{14}$$

$$\frac{\Gamma \vdash \tau \rhd \mathbf{Ok} \quad \Gamma \vdash \tau' \rhd \mathbf{Ok}}{\Gamma \vdash \tau \to \tau' \rhd \mathbf{Ok}} \tag{15}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \#C \in \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma)}{\Gamma \vdash \#C \rhd \mathbf{Ok}} \tag{16}$$

$$\frac{\mathit{\Gamma} \vdash \mathbf{Ok} \quad \forall m \in \mathcal{M}\ \mathit{\Gamma} \cup \{\alpha\} \vdash \tau_m \triangleright \mathbf{Ok}}{\mathit{\Gamma} \vdash \mathtt{obj}\,\alpha\,.\,\{\!|\, m : {\tau_m}^{m \in \mathcal{M}} \,|\!\} \triangleright \mathbf{Ok}} \tag{17}$$

$$\frac{\mathit{\Gamma} \vdash \tau \triangleright \mathbf{Ok} \quad \mathit{\Gamma} \vdash \#B \triangleright \mathbf{Ok} \quad \forall m \in \mathcal{M}\ \mathit{\Gamma} \vdash \tau_m \triangleright \mathbf{Ok}}{\mathit{\Gamma} \vdash (\tau)\,\{\!|\, \mathtt{inherits}\, B;\ m : {\tau_m}^{m \in \mathcal{M}} \,|\!\} \triangleright \mathbf{Ok}} \tag{18}$$

$$\frac{\forall C \in \mathrm{dom}(H) \quad \mathcal{H}(C) \in \mathrm{dom}(H) \cup \{\mathbf{None}\}}{H \vdash \mathbf{Ok}} \tag{19}$$

$$\frac{\begin{array}{c} \forall \tau \in \mathrm{rng}(\mathcal{S}\,\mathrm{of}\,\mathit{\Gamma})\ \mathit{\Gamma} \vdash \tau \triangleright \mathbf{Ok} \\ \forall \sigma \in \mathrm{rng}(\mathcal{C}\,\mathrm{of}\,\mathit{\Gamma})\ \mathit{\Gamma} \vdash \sigma \triangleright \mathbf{Ok} \\ \forall \tau' \in \mathrm{rng}(\mathcal{V}\,\mathrm{of}\,\mathit{\Gamma})\ \mathit{\Gamma} \vdash \tau' \triangleright \mathbf{Ok} \end{array}}{\mathit{\Gamma} \vdash \mathbf{Ok}} \tag{20}$$

C.4 Equality Judgements

$$\frac{\mathit{\Gamma} \vdash \tau \triangleright \mathbf{Ok}}{\mathit{\Gamma} \vdash \tau = \tau} \tag{21}$$

$$\frac{\mathit{\Gamma} \vdash \tau' = \tau}{\mathit{\Gamma} \vdash \tau = \tau'} \tag{22}$$

$$\frac{\mathit{\Gamma} \vdash \tau = \tau' \quad \mathit{\Gamma} \vdash \tau' = \tau''}{\mathit{\Gamma} \vdash \tau = \tau''} \tag{23}$$

$$\frac{\mathit{\Gamma} \vdash \tau_1 = \tau_2 \quad \mathit{\Gamma} \vdash \tau_1' = \tau_2'}{\mathit{\Gamma} \vdash \tau_1 \to \tau_1' = \tau_2 \to \tau_2'} \tag{24}$$

$$\frac{\mathcal{M}' = \mathcal{M} \quad \forall m \in \mathcal{M}\ \mathit{\Gamma} \cup \{\alpha\} \vdash \tau_m = \tau_m'}{\mathit{\Gamma} \vdash \mathtt{obj}\,\alpha\,.\,\{\!|\, m : {\tau_m}^{m \in \mathcal{M}} \,|\!\} = \mathtt{obj}\,\alpha\,.\,\{\!|\, m : {\tau'_m}^{m \in \mathcal{M}'} \,|\!\}} \tag{25}$$

$$\frac{\mathit{\Gamma} \vdash \mathtt{obj}\,\alpha\,.\,\{\!|\, m : {\tau_m}^{m \in \mathcal{M}} \,|\!\} \triangleright \mathbf{Ok}}{\begin{array}{l} \mathit{\Gamma} \vdash \mathtt{obj}\,\alpha\,.\,\{\!|\, m : {\tau_m}^{m \in \mathcal{M}} \,|\!\} \\ \qquad = \mathtt{obj}\,\alpha\,.\,\{\!|\, m : \tau_m[\alpha \mapsto \mathtt{obj}\,\alpha\,.\,\{\!|\, m : {\tau_m}^{m \in \mathcal{M}} \,|\!\}]^{m \in \mathcal{M}} \,|\!\} \end{array}} \tag{26}$$

C.5 Class Hierarchy Judgements

$$\frac{B \in \mathrm{dom}(H) \cup \{\mathbf{None}\}}{H \vdash B \lessdot B} \tag{27}$$

$$\frac{C \in \mathrm{dom}(H)}{H \vdash C \lessdot \mathbf{None}} \tag{28}$$

$$\frac{H \vdash B_1 \lessdot B_2 \qquad H \vdash B_2 \lessdot B_3}{H \vdash B_1 \lessdot B_3} \tag{29}$$

$$\frac{H(C) = B}{H \vdash C \lessdot B} \tag{30}$$

C.6 Subtyping Judgements

$$\frac{\Gamma \vdash \tau = \tau'}{\Gamma \vdash \tau <: \tau'} \tag{31}$$

$$\frac{\Gamma \vdash \tau <: \tau' \quad \Gamma \vdash \tau' <: \tau''}{\Gamma \vdash \tau <: \tau''} \tag{32}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \alpha \in \mathrm{dom}(\mathcal{S}\,\mathrm{of}\,\Gamma)}{\Gamma \vdash \alpha <: \mathcal{S}(\alpha)} \tag{33}$$

$$\frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma \vdash \tau_1' <: \tau_2'}{\Gamma \vdash \tau_1 \to \tau_1' <: \tau_2 \to \tau_2'} \tag{34}$$

$$\frac{\begin{array}{c} \mathcal{M}' \subseteq \mathcal{M} \quad \alpha' \notin \mathrm{FV}(\mathtt{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}) \\ \forall m \in \mathcal{M}'\ \alpha \notin \mathrm{FV}(\tau_m') \text{ and } (\Gamma \cup \{\alpha'\}) \pm \{\alpha \mapsto \alpha'\} \vdash \tau_m <: \tau_m' \end{array}}{\Gamma \vdash \mathtt{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} <: \mathtt{obj}\,\alpha'\,.\,\{\!|\, m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\}} \tag{35}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!|\, \mathtt{inherits}\,B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}}{\Gamma \vdash \mathtt{\#}C <: \mathtt{obj}\,\alpha\,.\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}} \tag{36}$$

$$\frac{\begin{array}{c} \Gamma \vdash \tau' <: \tau \quad H \vdash B \lessdot B' \quad \mathcal{M}' \subseteq \mathcal{M} \quad \forall m \in \mathcal{M}'\ \Gamma \vdash \tau_m = \tau_m' \\ \forall m \in (\mathcal{M} \setminus \mathcal{M}')\ \Gamma \vdash \tau_m \rhd \mathbf{Ok} \end{array}}{\Gamma \vdash (\tau)\,\{\!|\, \mathtt{inherits}\,B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} <: (\tau')\,\{\!|\, \mathtt{inherits}\,B';\ m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\}} \tag{37}$$

C.7 Typing Judgements

As a notational convenience, we define the argument type of a class C in an environment Γ (written $\mathrm{ArgTy}(\Gamma, C)$) to be τ if $\Gamma(C) = (\tau)\,\{\!|\, \mathtt{inherits}\,B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}$.

$$\frac{\Gamma \vdash d \rhd \Gamma' \quad \Gamma' \vdash p \rhd \Gamma'', \tau}{\Gamma \vdash d;\ p \rhd \Gamma'', \tau} \tag{38}$$

$$\frac{\Gamma \vdash e \rhd \tau}{\Gamma \vdash e \rhd \Gamma, \tau} \tag{39}$$

$$\frac{\begin{array}{c} C \notin \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma) \quad \Gamma \vdash \tau \rhd \mathbf{Ok} \\ \Gamma' = \Gamma \pm \{C \mapsto (\tau)\,\{\!|\, \mathtt{inherits}\,\mathbf{None};\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}\} \\ \forall m \in \mathcal{M}\ \Gamma' \pm \{\mathit{self} \mapsto \mathtt{\#}C\} \vdash \mu_m \rhd \tau_m' \qquad \Gamma' \vdash \tau_m' <: \tau_m \end{array}}{\Gamma \vdash \mathtt{class}\ C\ (x : \tau)\,\{\!|\ m = \mu_m{}^{m \in \mathcal{M}} \,|\!\} \rhd \Gamma'} \tag{40}$$

$$\frac{\begin{array}{c} C \notin \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma) \\ (\mathcal{C}\,\mathrm{of}\,\Gamma)(C') = (\tau')\,\{\!|\, \mathtt{inherits}\,B';\ m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\} \\ \Gamma' = \Gamma \pm \{C \mapsto (\tau)\,\{\!|\, \mathtt{inherits}\,C';\ m : \tau'_m{}^{m \in (\mathcal{M}' \setminus \mathcal{M})}\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}\} \\ \Gamma' \pm \{x \mapsto \tau\} \vdash e \rhd \tau'' \quad \Gamma' \vdash \tau'' <: \tau' \\ \forall m \in (\mathcal{M} \cap \mathcal{M}')\ \Gamma' \vdash \tau_m <: \tau_m' \\ \forall m \in \mathcal{M}\ \Gamma' \pm \{\mathit{self} \mapsto \mathtt{\#}C\} \vdash \mu_m \rhd \tau_m'' \qquad \Gamma' \vdash \tau_m'' <: \tau_m \end{array}}{\Gamma \vdash \mathtt{class}\ C\ (x : \tau)\,\{\!|\, \mathtt{inherits}\,C'(e);\ m = \mu_m{}^{m \in \mathcal{M}} \,|\!\} \rhd \Gamma'} \tag{41}$$

$$\frac{C \notin \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma) \quad \Gamma' = \Gamma \pm \{C \mapsto \sigma\} \quad \Gamma' \vdash (\mathcal{C}\,\mathrm{of}\,\Gamma)(C') <: \sigma}{\Gamma \vdash \mathtt{class}\ C : \sigma = C' \rhd \Gamma'} \tag{42}$$

$$\frac{\Gamma \pm \{x \mapsto \tau\} \vdash e \rhd \tau'}{\Gamma \vdash (x\ :\ \tau) \Rightarrow e \rhd \tau \to \tau'} \tag{43}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{V}\,\mathrm{of}\,\Gamma)(x) = \tau}{\Gamma \vdash x \rhd \tau} \tag{44}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{V}\,\mathrm{of}\,\Gamma)(\mathtt{self}) = \tau}{\Gamma \vdash \mathtt{self} \rhd \tau} \tag{45}$$

$$\frac{\Gamma \pm \{x \mapsto \tau\} \vdash e \rhd \tau'}{\Gamma \vdash \mathtt{fn}(x\ :\ \tau) \Rightarrow e \rhd \tau \to \tau'} \tag{46}$$

$$\frac{\Gamma \vdash e \rhd \tau' \to \tau \quad \Gamma \vdash e' \rhd \tau'' \quad \Gamma \vdash \tau'' <: \tau'}{\Gamma \vdash e(e') \rhd \tau} \tag{47}$$

$$\frac{\Gamma \vdash e \rhd \tau' \quad \Gamma \vdash \tau' <: \mathrm{ArgTy}(\Gamma,\, C)}{\Gamma \vdash \mathtt{new}\ C(e) \rhd \mathtt{\#}C} \tag{48}$$

$$\frac{\Gamma \vdash e \rhd \mathtt{\#}C}{\Gamma \vdash e\mathtt{!state} \rhd \mathrm{ArgTy}(\Gamma,\, C)} \tag{49}$$

$$\frac{\Gamma \vdash e \rhd \mathtt{\#}C' \quad \mathcal{H}(\Gamma) \vdash C' \lessdot C}{\Gamma \vdash e\,@\,C \rhd \mathtt{\#}C} \tag{50}$$

$$\frac{\Gamma \vdash e \rhd \tau \quad m \in \mathcal{M} \quad \Gamma \vdash \tau <: \mathtt{obj}\,\alpha\,.\{\!|\, m : {\tau_m}^{m \in \mathcal{M}} \,|\!\}}{\Gamma \vdash e\,.\,m \rhd \tau_m[\alpha \mapsto \mathtt{obj}\,\alpha\,.\{\!|\, m : {\tau_m}^{m \in \mathcal{M}} \,|\!\}]} \tag{51}$$

C.8 Typing Rules for Run-Time Forms

The typing rules for the run-time forms include rules 43 through 51 augmented with a dynamic class hierarchy H on the left-hand side of the turnstile. Rules 40 through 42 return an augmented class hierarchy in addition to an augmented static environment. In each case, the augmented hierarchy is just the old one extended with a mapping from the newly declared class name to the name of its parent (or **None** for base classes). In addition, rule 50 is replaced by rule 55. The remaining rules are given below.

$$\frac{\Gamma, H \vdash e \rhd \tau' \quad \Gamma \vdash \tau' <: \mathrm{ArgTy}(\Gamma,\, C)}{\Gamma, H \vdash C(e) \rhd \mathtt{\#}C} \tag{52}$$

$$\frac{H(C) = B \quad \Gamma, H \vdash \mathit{obj} \rhd \mathtt{\#}B \quad \Gamma, H \vdash e \rhd \tau' \quad \Gamma \vdash \tau' <: \mathrm{ArgTy}(\Gamma,\, C)}{\Gamma, H \vdash C :: \{\!|\, e;\ \mathit{obj} \,|\!\} \rhd \mathtt{\#}C} \tag{53}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \mathcal{K} \vdash \mathbf{Ok}}{\Gamma, H \vdash \mathbf{None} \rhd \mathtt{\#}\mathbf{None}} \tag{54}$$

$$\frac{\mathit{\Gamma}, H \vdash e \triangleright \texttt{\#}C' \quad H \vdash C' \lessdot C}{\mathit{\Gamma}, H \vdash e\,@\,C \triangleright \texttt{\#}C} \tag{55}$$

$$\frac{\mathit{\Gamma}, H \vdash \mathit{obj} \triangleright \texttt{\#}C' \quad H \vdash C' \lessdot C}{\mathit{\Gamma}, H \vdash \langle \mathit{obj},\ C \rangle \triangleright \texttt{\#}C} \tag{56}$$

$$\frac{\begin{array}{c}
\sigma = (\tau)\,\{\!\!|\,\texttt{inherits}\,\mathbf{None};\ m : {\tau_m}^{m \in \mathcal{M}}\,|\!\!\} \\
\mathcal{M} = \mathrm{dom}(\phi) \qquad \mathrm{rng}(\phi) \subseteq \mathrm{dom}(\mu T) \\
\forall m \in\ \mathcal{M}\ \mathit{\Gamma} \pm \{\mathit{self} \mapsto \texttt{\#}C\}, H \vdash \mu T(\phi(m)) \triangleright \tau'_m \\
\mathit{\Gamma} \vdash \tau'_m <: \tau_m
\end{array}}{\mathit{\Gamma}, H, C \vdash (\mathbf{None},\ ??,\ \mu T,\ \phi) \triangleright \sigma} \tag{57}$$

$$\frac{\begin{array}{c}
\sigma = (\tau)\,\{\!\!|\,\texttt{inherits}\,C';\ m : {\tau_m}^{m \in \mathcal{M}}\,|\!\!\} \\
\mathit{\Gamma}(C') = (\tau')\,\{\!\!|\,\texttt{inherits}\,B';\ m : {\tau'}_m{}^{m \in \mathcal{M}'}\,|\!\!\} \\
\mathit{\Gamma}, H \vdash e \triangleright \tau \to \tau'' \quad \mathit{\Gamma} \vdash \tau'' <: \tau' \\
\mathcal{M} = \mathrm{dom}(\phi) \qquad \mathrm{rng}(\phi) \subseteq \mathrm{dom}(\mu T) \\
\forall m \in\ \mathcal{M}\ \mathit{\Gamma} \pm \{\mathit{self} \mapsto \texttt{\#}C\}, H \vdash \mu T(\phi(m)) \triangleright \tau''_m \\
\mathit{\Gamma} \vdash \tau''_m <: \tau_m \\
\forall m \in\ \mathcal{M} \cap \mathcal{M}'\ \mathit{\Gamma} \vdash \tau_m <: \tau'_m
\end{array}}{\mathit{\Gamma}, H, C \vdash (C',\ e,\ \mu T,\ \phi) \triangleright \sigma} \tag{58}$$

$$\frac{\begin{array}{c}
\mathrm{dom}(\mathcal{K}) = \mathrm{dom}(\mathcal{C}\ \text{of}\ \mathit{\Gamma}) \\
\forall C \in \mathrm{dom}(\mathcal{K})\ \ \mathit{\Gamma}(C) = \sigma\ \text{and}\ \mathit{\Gamma}, H(\mathcal{K}), C \vdash \mathcal{K}(C) \triangleright \sigma'\ \text{and}\ \mathit{\Gamma} \vdash \sigma' <: \sigma
\end{array}}{\mathit{\Gamma} \vdash \mathcal{K}} \tag{59}$$

A Basic Model of Typed Components

João Costa Seco and Luís Caires

Departamento de Informática - Universidade Nova de Lisboa
{Joao.Seco,Luis.Caires}@di.fct.unl.pt

Abstract. This paper studies specific language level abstractions for component-based programming. We propose a simple model which captures some basic ingredients — like explicit context dependence, dynamic binding and subtype polymorphism, late (dynamic) composition, and avoidance of inheritance in favor of object composition — that several authors have defended to be central to black-box object-oriented component programming styles. The model is expressed by a core imperative typed calculus, in which components are first-class entities, and whose basic constructs enable the composition, scripting, instantiation and definition of atomic components. Some motivating programming examples are presented, and the operational semantics is shown to enjoy a type-safety property. We also discuss an extension to the Java language aimed at supporting the proposed model, and some implementation issues.

1 Introduction

The development of tools and techniques for component-based programming (COP) is a subject which receives increasing interest in recent years. Industrial strength component models continue to be proposed and refined, while fundamental research is engaged into clarifying the basic concepts involved. An important issue concerns the study of specific language level abstractions for COP, since the most popular supporting models [21,14,20] resort to low-level idioms and do not enforce type-safety of composition. As with OOP, a major promise of COP is related to software reuse.

Object-oriented programming languages promote reuse relying on classes, inheritance and polymorphism. Given some class framework, new classes can be defined that subclass one (or more) of previously defined classes and automatically inherit concrete elements from them, like state variables and method implementations, which can then be overridden and extended. This methodology of providing reusable classes relies on a form of implicit parametricity. In some cases, the superclass can be abstract and provide no default implementation for some methods intended to be defined in a subclass, abstract methods are therefore similar to bottleneck interfaces.

This approach to reuse, implicit in usual object-oriented languages, has been subject to criticism with relation to its ability to provide a convenient support for COP [23,26]. Usually subclasses are defined w.r.t. some preexisting fixed super-classes, assuming a globally defined and well-known framework. In general,

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 108–128, 2000.
© Springer-Verlag Berlin Heidelberg 2000

super-classes cannot be maintained separately from their derived classes, since the implementation of the latter depends in general on internal implementation details and interface of the former — the so-called fragile base class problem [11]. In other words, the (undisciplined) presence of inheritance may inhibit a desirable degree of encapsulation needed for independent maintenance and modifiability [19]. On the other hand, in order to be reusable and independently deployed in an effective way, a component must interact with its environment only through explicitly defined interfaces, expressing contracts between service providers and clients.

The rigid anchoring of subclasses on super-classes may be weakened by decoupling the explicit sub-classing operation from the definition of the superclass itself. This separation of class definition and sub-classing is accomplished by the notion of "mixin". Mixins [4,3,10,2] are essentially class defining functions which are explicitly parametric on their superclass. A mixin defines a particular sub-classing operation which can be applied to several different classes conforming with a given specialization interface, to yield several particular subclasses: elementary mixin operations include, for instance, addition and overriding of methods.

Nevertheless, the semantics of mixins is still defined such that inheritance with late binding of self is still ensured through mixin application. Several proposals have already been put forward to discipline the inheritance mechanism [17,25], but it is still true that sub-classing — either based on explicit sub-classing or relying on mixins — addresses reuse by modifying the inherited code (even if such modifications are limited to method overriding) rather than reusing it as a whole [26]. The cumulative effects of method overriding along subclass chains are in general hard to specify due to the intrinsic recursive nature of self invocation under dynamic binding. Because of this, several researchers defended that inheritance "as-normally-used" should be highly disciplined or even forbidden across components borders [19,26,23]. This discussion also applies to delegation-based (class-less) object models [24], which also provide implementation inheritance with late binding of self.

Another related issue concerns reuse of code from several sources by means of multiple inheritance, which is in general a complex and not very popular mechanism, although available in widespread OO programming languages. On the other hand, the idea of aggregation of multiple components is very natural, and does not suffer from the problems presented by multiple inheritance.

This paper studies specific language constructs to support COP, in the setting of inheritance-free object-oriented programming within a standard (Java-like) imperative semantics. The technical contribution of this paper is the proposal of a simple core model which captures some basic ingredients essential to black-box object-oriented component programming styles like explicit context dependence, dynamic binding and subtype polymorphism, multiple views, late (dynamic) composition, and avoidance of inheritance in favor of object composition. Components are treated as closed first-class values of component type. This fact reflects, in an abstract way, that components are stateless self-contained units open to late composition and instantiation, and able to be dynamically loaded, instantiated [18]) and possibly transmitted across communication channels.

The model is expressed by a core imperative typed formal calculus, in which components are first-class entities, and whose basic constructs enable the composition, scripting, instantiation and definition of atomic components.

Structure of the paper. After giving a intuitive motivation of the component definition and composition operations using an extension to the Java programming language named ComponentJ (Sec.2), we present the core component calculus, its type-system and operational semantics (Sec.3). This section is concluded by the statement of type-safety property. In Sec.4 we make some preliminary remarks about the implementation of ComponentJ. The paper ends with a discussion of related work and some conclusions.

2 Motivating Sketch

In this section, we motivate the basic features of our model of typed components. We pick as a running example a component implementing a personal to-do list, based on a queue of generic tasks. Our task manager provides two distinct views of the queue: a "user" view that lets an user enqueue or dequeue some task, which is defined by the interface `IQueue` and a "supervisor" view that allows certain privileged operations defined by the interface `ISupervisor`, both defined in Fig.1.

A possible implementation of the queue can rely, say, on a linked list data structure. Therefore, we start by defining a component named `CList` implementing the `IList` interface. The component declaration is made by the `component` block, containing a set of declarations composition operations that define the component structure, its required and provided services. The general syntax is illustrated by the examples in Fig.2. The `CList` component provides a service implementing interface `IList` at a port named `list`; this is expressed by the `provides` clause. The list itself is implemented by the block of methods `m`, which is made available to the outside through a plugging operation. The `plug` operation has the effect of connecting the method block `m` to the port `list`. In general,

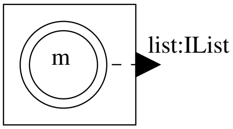
```
interface IQueue {
  void enQueue(ITask t);
  ITask deQueue();
}

interface IList {
  void insert(int p,Object o);
  void remove(int p);
  Object get(int p);
  int size();
}
```

```
interface ISupervisor {
  int size();
  ITask get(int p);
  void remove(int p);
  void swap(int p1, int p2);
  void visit(IVisitor v);
}
```

Fig. 1. Interfaces of the `CToDo` component

```
component CList is compose {
  provides IList list;
  meth m {
    Object elements[];
    void insert(int p,Object o) {..}
    ...
  }
  plug m into list;
}
```

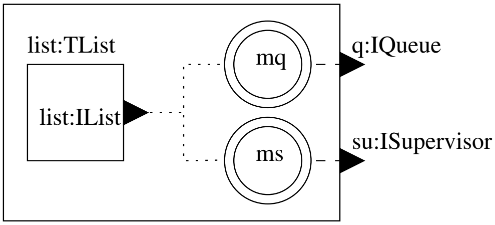
(a)



(b)

```
component CToDo is compose {
  provides IQueue q;
  provides ISupervisor su;
  intro CList list;
  meth mq {
    void enQueue(ITask t) {..}
    ITask deQueue() {..}
  }
  meth ms {
    int size() {..}
    ...
  }
  plug mq into q; plug ms into su;
}
```

(c)



(d)

Fig. 2. The `CList` and `CToDo` components.

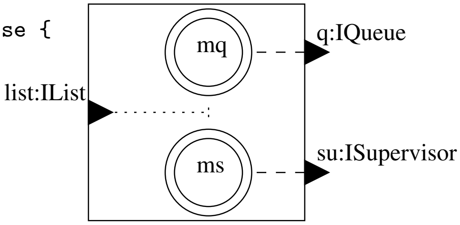
a plug operation can connect a source port to a destination port whenever the source's interface type is a subtype of the destination's interface type.

From now on, we will graphically depict components as in Fig.2(b), where squares represent components, double circles represent method blocks, and filled triangles represent component interfaces. With a dashed line we represent the result of a plugging operation.

Now, given the component implementing the `IList` interface, the basic `CToDo` component can be defined as shown in Fig.2(c). The `intro` clause introduces a `CList` internal component, whose service ports are implicitly visible in the method blocks' scope. Dotted lines represent the implicit use of local names in the code of method blocks (e.g `l` is visible inside the methods of `mq`). To implement the `CToDo` component, two method blocks `mq` and `ms` are defined based on the inner component's `list` service, and then plugged into the corresponding provided interfaces through the `plug` clauses.

```
component CToDoExtension is compose {
  provides IQueue q;
  provides ISupervisor su;
  requires IList list;
  meth mq {
    void enQueue(ITask t) {..}
    ITask deQueue() {..}
  }
  meth ms { int size() {..}
  ...
  }
  plug mq into q; plug ms into su;
}
```

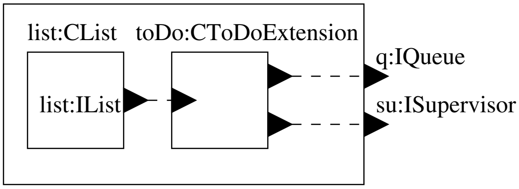
(a)



(b)

```
component CToDoModular
  is compose {
    provides IQueue q;
    provides ISupervisor su;
    intro CToDoExtension toDo;
    intro CList list;
    plug list.list into toDo.list;
    plug toDo.q into q;
    plug toDo.su into su;
}
```

(c)



(d)

Fig. 3. The `CToDoExtension` and `CToDoModular` components.

The existence of several service interfaces in a component instance is useful to cope with several different views of a component. A major difference between this approach and the implementation of multiple interfaces like in Java, or multiple inheritance in C++, is that it is not possible to cast from one interface type to another. The same observation applies to compound types [5].

Factoring Out. In our implementation of the `CToDo` component, the `CList` component is hardwired inside. However, this dependency can be made explicit, allowing for a extra potential of reuse. The resulting component can then be used with any other component implementing the `IList` interface: say a persistent list or a remote list. To that end, we factor out the component implementing the list and make the dependency of the to-do list on a list service explicit through a required service interface. This architectural refinement step yields the `CToDoExtension` component defined in Fig.3(a). This new component can then be composed in the `CToDoModular` component like in Fig.3(c) with the functionality of of the `CToDo` component.

```
interface ISelector {
  int select();
}

interface TSelectableToDo
  provides IQueue q
  provides ISupervisor su
  requires ISelector s
  requires IList list;

interface TOrder
  provides ISelector s
  requires IList l;

interface TToDo
  provides IQueue q
  provides ISupervisor su;
```

(a)

```
component CNewToDo is compose {
  provides IQueue q;
  provides ISupervisor su;
  intro compose {
    provides IQueue q;
    provides ISupervisor su;
    requires IList list;
    intro COrder policy;
    intro CSelectableToDo toDo;
    plug list into policy.list;
    plug policy.s into toDo.s;
    plug list into toDo.list;
    plug toDo.q into q;
    plug toDo.su into su;
  } toDoExtension;
  intro CList list;
  plug list.list into toDoExtension.list;
  plug toDoExtension.q into q;
  plug toDoExtension.su into su;
}
```

(b)

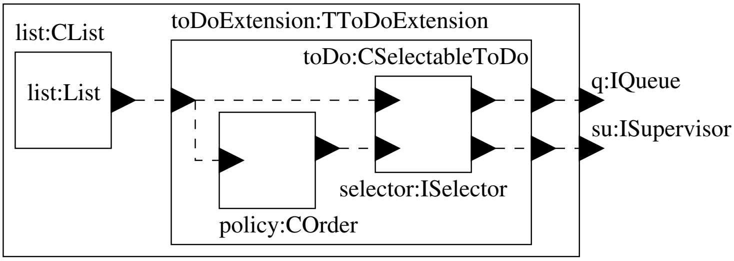


Fig. 4. A further composition example

Replacing Components. Now, suppose that a component CSelectableToDo is released by some software house providing and requiring the same services as our CToDoExtension, plus an additional required service that allows for the configuration of an arbitrary selection policy over the queue. The interface type of this service is the ISelector interface type shown in Fig.4(a). This component's type is the TSelectableToDo also shown in Fig.4(a). Assigning component types to components allows static type-checking of component composition operations to be performed. Component types specify a set of typed required services and a set of provided services. Note that component types are distinct from component instance types. Instance types are syntactically similar to component types with an empty set of requirements, however, an instance is a service provider (similar to an object), while a component is a unit composition that can be composed and instantiated to yield component instances (like a class). Thus, there is a clear separation between the generating entities and the types of the generated instances (e.g components are not types).

Therefore, in order to replace CToDoExtension in our CToDoModular component in Fig.3(c), we need to provide an implementation of the ISelector interface, relying on the contents of a list. This is accomplished by defining a custom component COrder. This component, not shown here, must conform with the component type TOrder, which is declared in Fig.4(a). The composition of the COrder component with the CSelectableToDo yields a component that is able to replace the original CToDoExtension, as shown in Fig.4(b) yielding the CNewToDo component. This new component can replace the original CToDo component in any context because it has the same type, namely TToDo as defined in Fig.4(a). We adopt a natural notion of sub-typing among component types, which will be described in the next Section.

Dynamic Composition. Since we take components as first-class citizens, methods can be defined to return component values. Therefore, we can define in some component a factory method makeToDo that returns a dynamically linked component of TToDo, based on a given component with the TOrder type, as shown below

```
TToDo makeToDo(TOrder customOrder) {
  return compose {
            ...
            intro compose {
               ...
               intro customOrder policy;
               ...
            } toDoExtension;
            ...
         }
}
```

This example illustrates a simple case of dynamic composition. Note that the parameter customOrder occurs free in the compose expression in the body of method makeToDo. In fact, only variables of component type can occur free in component expressions. Typically, such variables denote parameters that will be bound to closed component values.

Instantiating Components. Client code can instantiate components with the new operator, like for classes in Java. Components without required services can be instantiated in a similar way to regular classes in OO languages. In the example of Fig.5, a new component instance, of type ITToDo, is obtained by instantiating the CNewToDo component. On the other hand, if a reference to an object or service port of a component that implements the IList interface is available, it can be used at instantiation time to satisfy the pending requirement of a component like CToDo. Note that at instantiation time all requirements of a component must be resolved, this property is statically ensured by the type-system.

```
interface ITToDo implements TToDo;

class Test {
  main() {
    ITToDo tm = new CNewToDo();
    tm.queue.enQueue(new Task("...", new Date(1999,12,20)));
    tm = new CToDoExtension() {plug new DList() into list;};
    ...
  }
}
```

Fig. 5. Instantiation of a component

3 The Core Calculus of Components

In this section we present the syntax and semantics of the core calculus of components. The calculus is explicitly typed, so we start by introducing the required type structure and then define the terms of the calculus.

3.1 Types

The type structure of the core calculus is defined from a few basic kinds of types: *interface types* (I), *component types* ($R \Rightarrow P$), *method types* (ι) and *instance types* $[P]$; the abstract syntax of these types is presented in Fig.6. We assume given sets of countably many method ($f_1, \dots$) and port ($p_1, \dots$) names. Interface types I are used to type ports of components and method blocks. Each one specifies a set of method signatures by assigning to each method name f_i a method type ι_i of the form $(\tau_1, \dots, \tau_m)\sigma$. In such a method type, the τ_i specify the types of the method's parameters and σ the return type, as expected.

We assume given a globally defined set $\mathcal{N}$ of *interface names* and a mapping $\mathcal{I}$ assigning an interface type to each interface name. The motivation for interface names is more to allow for a technically simpler introduction of recursive interface types, rather than to introduce name equivalence of types. In fact, for the sake of generality, the calculus presented in this paper assumes structural equivalence of types, unlike our proposed Java extension ComponentJ, which adopts name

$$
\begin{array}{lll}
\tau, \sigma & ::= I \mid R \Rightarrow P \mid [R] & \\
R, P & ::= \{p_1 : I_1, \dots, p_n : I_n\} & (n \geq 0) \\
\iota & ::= (\tau_1, \dots, \tau_n)\sigma & (n \geq 0) \\
I & ::= \mathtt{I} \in \mathcal{N} \mid \{f_1 : \iota_1, \dots, f_n : \iota_n\} & (n \geq 0)
\end{array}
$$

Fig. 6. Component Types.

equivalence of interface types and an explicitly given relation of interface subtyping.

A type $R \Rightarrow P$ types components that require a *service set* as specified by R and provide a service set as specified by P; each service is denoted by an interface type tagged by a port name. Service sets have the form $\{p_1 : I_1, \ldots , p_n : I_n\}$, where each p_i is a port name and I_i an interface type. Thus, components cannot "import" other components *per se*, but just access some services provided by other components. Nevertheless, the calculus allows for the definition of parametric component factories, that is, the assembly of components from dynamically determined basic constituents. When R is a service set like the one above, we write $R(p_i)$ for the interface type assigned to the service port p_i by R. Components compose to yield instantiable components — as with classes — which are essentially components with an empty set of requirements. Therefore, component instances providing a service set R will be typed $[R]$.

3.2 Terms

The terms of the calculus are described by the abstract syntax in Fig.7. The first seven forms of expression embody the computational core of the language: we have *variable* x, *port selection* $e.p$, *method call* $e.f(\boldsymbol{e})$, *assignment* $x := e$, *dereference* $!x$, *local definition* $\mathsf{let} - \mathsf{in}$, *composition* $\mathsf{compose}\ e$ and *instantiation* $\mathsf{new}-\mathsf{with}-\mathsf{in}$. Heap locations are represented by the constants of the form ℓ and $\ell \in \mathsf{Loc}$ that denotes heap locations. Additionally a special value nil is also used to represent unitialized locations. A error constant is used to represent wrong computations. Note that heap locations, nil and error are never used in source programs, but just instrumental in the definition of the operational semantics. Some abbreviations will be used throughout the paper in order to simplify the reading. We will write $e_1; e_2$ as an abbreviation for $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, as usual; and just $\mathsf{new}\ e$ when $\boldsymbol{e}$ and $\boldsymbol{p}$ are void. Other abbreviations will be pointed out as needed. We refrain from introducing additional forms for the sake of simplicity; the given constructs are enough to capture the intended basic imperative model. For example, we do not introduce component initializers similar to constructors in classes, which could be easily added on top of the present core. Most of the constructs have the usual meaning, except composition $\mathsf{compose}\ e$, which denotes a component as specified by e, and instantiation $\mathsf{new}\ e\ \mathsf{with}\ \boldsymbol{e}\ \mathsf{in}\ \boldsymbol{p}$. In such an expression, e must evaluate to a component with requirements $\{\boldsymbol{p} : \boldsymbol{I}\}$. The value of such an expression is a new instance of the component specified by e and obtained by plugging the value of each expression e_i into the corresponding requirement p_i; the port names $\boldsymbol{p}$ must be pairwise distinct.

The remaining forms are used to define compound components by gluing together other components, and by scripting primitive behavior with method blocks. $\boldsymbol{\varepsilon}$ denotes the *empty component.* The expression $\mathsf{require}\ x : I\ \mathsf{and}\ e$ denotes a component that requires a service of type I at port x and is further specified by e; inside e the port name x is available for service calls, and for plugging into compatible destination. As we shall see later, all names visible inside a component definition must be either explicitly imported from the outer

$$\begin{array}{rl}
e ::= & x \mid e.p \mid e.f(\boldsymbol{e}) \mid {!x} \mid x := e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid \\
& \mathsf{compose}\ e \mid \mathsf{new}\ e\ \mathsf{with}\ \boldsymbol{e}\ \mathsf{in}\ \boldsymbol{p} \mid \boldsymbol{\varepsilon} \mid \\
& \mathsf{require}\ x : I\ \mathsf{and}\ e \mid \mathsf{provide}\ x : I\ \mathsf{and}\ e \mid \\
& \mathsf{intro}\ x : C = e\ \mathsf{and}\ e \mid \mathsf{meth}\ x : \mathcal{M}\ \mathsf{and}\ e \mid \\
& \mathsf{plug}\ \pi\ \mathsf{into}\ \pi\ \mathsf{and}\ e \mid \ell \mid \mathsf{nil} \mid \mathsf{error}
\end{array}$$

$$\pi ::= p \mid x.p \qquad\qquad \mathcal{M} ::= \{a_1 : \tau_1, \ldots ; f_1(\boldsymbol{x}) = e_1, \ldots\}$$

Fig. 7. Core terms.

environment through required service ports or made available by internally introduced components — components are always closed entities. Expression form provides $x : I$ and e denotes a component that provides a service of type I at port x and is further specified by e; in particular, e must specify a connection of some internally available compatible source to this port name x. We will require all port names of a component to be distinct.

The construct intro $x : C = e_1$ and e_2 introduces an inner component for assembly which will be referred inside e_2 by the variable x, which is locally bound and has e_2 as its scope. Such inner component is specified by e_1 and must conform with the component type C. The body e_2 must specify how its requirements are to be satisfied by internally available services. Plugging of available services — either provided by inner components or imported from the external environment through requirement ports — to suitable destinations is accomplished by the expression form plug π_1 into π_2 and e. A *plug* π is either a port name p of the component being assembled, or of the form $x.p$, where x refers to some inner component and p a name of a port of x. Primitive behavior and scripting of predefined components is obtained by a method block construct meth $x : \mathcal{M}$ and e; again the name x is local and bound within scope e. Method blocks encapsulate both local state variables and method definitions. Upon instantiation they introduce object-like entities that "implement" the interface defined by the signature of its methods. In a method definition such as $m(\boldsymbol{x} : \boldsymbol{\tau}) = b$, the parameters $\boldsymbol{x}$ are local binding occurrences and the method body b can directly call any

$$\begin{array}{l}
\mathsf{require}\ l : \mathit{List}\ \mathsf{and} \\
\mathsf{provide}\ su : \mathit{ISupervisor}\ \mathsf{and} \\
\mathsf{provide}\ q : \mathit{IQueue}\ \mathsf{and} \\
\mathsf{meth}\ mq : \{ \\
\quad \mathit{enqueue}(t : \mathit{Task}) = l.\mathit{insert}(l.\mathit{size}(), t), \\
\quad \mathit{dequeue}() = \mathsf{let}\ x = l.\mathit{get}(0)\ \mathsf{in}\ l.\mathit{remove}(0); x\}\ \mathsf{and} \\
\mathsf{meth}\ ms : \{\mathit{size}() = l.\mathit{size}(), \ldots\}\ \mathsf{and} \\
\mathsf{plug}\ mq\ \mathsf{into}\ q\ \mathsf{and} \\
\mathsf{plug}\ ms\ \mathsf{into}\ su\ \mathsf{and}\ \boldsymbol{\varepsilon}
\end{array}$$

Fig. 8. The `CToDoExtension` component.

service internally available in the component being built. In Fig.8 we present an example of a well-defined term of the calculus; this term corresponds to the previously given specification of component `CToDoExtension` given in Section 2.

Example. We show how code inheritance with dynamic binding can be simulated by explicit parameterization in the calculus. With this example we do not pretend to suggest a particularly natural component-oriented programming pattern, but rather to give a comparative illustration of expressiveness. Note that we have omitted " and ε" at the end of component expressions, for the sake of brevity. Let I be an interface as $I = \{do : \iota_{do}, twice : \iota_{twice}\}$, and let

$$\begin{aligned}&C = \mathsf{compose}(\mathsf{require}\ s : I\ \mathsf{and}\\&\mathsf{meth}\ m : \{do() = b_1, twice() = s.do(); s.do(); \}\\&\mathsf{and\ provide}\ o : I\ \mathsf{and\ plug}\ m\ \mathsf{into}\ o)\end{aligned}$$

C represents a "class" which enables overriding of methods in I, and that can be instantiated by

$$\begin{aligned}&\mathsf{new\ compose}(\mathsf{provide}\ o : I\ \mathsf{and\ intro}\ x : \tau_C = C\\&\ \mathsf{and\ plug}\ x.o\ \mathsf{into}\ x.s\ \mathsf{and\ plug}\ x.o\ \mathsf{into}\ o)\end{aligned}$$

Basically, the specialization interface (or alternatively, the set of abstract methods) is made explicit by a required interface $s : I$ of the component C representing a "base" class. Moreover, all calls to "self" made in bodies of exported methods are forwarded to s. We now extend "class" C by "overriding" method *do*. We can now define a "mixin"-like component

$$\begin{aligned}&M = \mathsf{compose}(\mathsf{require}\ s : I\ \mathsf{and}\\&\mathsf{meth}\ m : \{do() = b_2; s.do(), four() = s.twice(); s.twice()\}\\&\mathsf{and\ provide}\ o : J\ \mathsf{and\ join}\ s, m\ \mathsf{into}\ o)\end{aligned}$$

where $J = I \cup \{four : \iota_{four}\}$. Here we have used a join expression, which is an idiom of the core calculus reducible to meth and plug: join a, b into c just abbreviates a facade that forwards a method call $f(\boldsymbol{e})$ originating at o to m if the interface type of m defines f and to s otherwise. Hence, M redefines method $do()$ — reusing the code provided by C — and introduces a new method $four()$. Now M can be composed with C as in

$$\begin{aligned}&N = \mathsf{compose}(\mathsf{require}\ s : I\ \mathsf{and}\\&\quad \mathsf{intro}\ x : \tau_C = C\ \mathsf{and\ intro}\ y : \tau_M = M\ \mathsf{and\ plug}\ s\ \mathsf{into}\ x.s\ \mathsf{and}\\&\quad \mathsf{and\ plug}\ x.o\ \mathsf{into}\ y.s\ \mathsf{and\ provide}\ o : I\ \mathsf{and\ plug}\ y.o\ \mathsf{into}\ o))\end{aligned}$$

and instantiated as C above. For instance, in the program

$$\begin{aligned}&\mathsf{let}\ MI = \mathsf{compose}(\mathsf{provide}\ o : I\ \mathsf{and\ intro}\ x : \tau_N = N\ \mathsf{plug}\ x.o\ \mathsf{into}\ x.s\\&\qquad\qquad\qquad\quad \mathsf{and\ plug}\ x.o\ \mathsf{into}\ o)\\&\quad \mathsf{in\ let}\ obj = (\mathsf{new}\ MI).o\ \mathsf{in}\ obj.twice()\end{aligned}$$

the "inherited" code for *twice* will call the definition for $do()$ in M. Note that while N is still open to redefinition of methods declared in the required interface I, MI closes the cycle and is an instantiable component (of type $\emptyset \Rightarrow \{o : J\}$). Thus, a component like N plays a role similar to the object generating function used in many semantic models of object-oriented languages and mixins (for instance [2]), and which require a fixpoint operator to be applied at object instantiation time in order to resolve the involved circularity. Herein, self-reference is captured by the imperative reference semantics.

3.3 Type System

In this section, we present a type system for the core calculus of components. Well-typed programs do not incur in execution errors, in a sense to be made precise below. We start by defining a natural relation of sub-typing and write $\tau <: \sigma$ when τ is a subtype of σ. This relation is defined in the expected way; in Fig.9 we present all the sub-typing rules. In particular, a component type τ is a subtype of a component type σ whenever τ provides more services and requires less services than those specified by σ. Moreover, each service provided by τ must conform with an interface which is a subtype of the corresponding service provided by σ, and contravariantly for required services. This notion of sub-typing for components is the natural one, and has been adopted by many recent approaches to modularity [9,12], but also in [16] and already implicit in [6].

The type system is defined by relying on two judgment forms, which define typing of general expressions and components in a mutually recursive way. For general expressions, we have the standard judgment form $\Delta \vdash e : \tau$ where Δ is a typing context, e a term and τ a type. Typing contexts are plain assignments of types to variables: hence we write $\Delta(x) = \tau$ whenever $\Delta = \Delta', x : \tau$; and $\Delta(x.p) = I$ whenever $\Delta = \Delta', x : [p : J, \ldots]$. For components we introduce the judgment form $\mathcal{R}; \Delta \vdash_c e : \tau$ where Δ, e and τ play the usual role, while $\mathcal{R}$ represents a set of *internal requirements* generated by that e must resolve by means of pluggings. Each such requirement is an assignment of an interface type to some plug π. The full set of typing rules for general expressions is presented in Fig.9. For the strict use of the typing system, we introduce the additional kind of type $\mathsf{Var}(\sigma)$ for any type σ. Var types are never present in source terms, but are needed to type state variables introduced by method blocks. In the premise of the rule for $\mathsf{compose}\ e$, we introduce a subset $|\Delta|$ of the typing context Δ appearing in the conclusion, defined by $|\Delta| = \{x : C \in \Delta : C \text{ is a component type}\}$. This ensures that "code" inside the component e cannot access any global variables from the outside context — and is therefore a closed unit — except possibly for variables referring to other components. But such names will again denote closed components that could be explicitly linked inside e — say, in a concrete implementation — to yield a closed component. A consequence of this is that all dependences of a component on its environment must be brought explicit in its interfaces, unlike in [9,1] where modules can be open (contain free variables). The typing rules for component expressions are also presented in Fig.9. The set

Sub-typing

$$\tau <: \tau \qquad \frac{\tau_i <: \tau_i'}{\{f_1 : \tau_1, \ldots, f_n : \tau_n\} <: \{f_1 : \tau_1', \ldots, f_m : \tau_m'\}} \ (m \leq n)$$

$$\frac{\sigma_i <: \tau_i \quad \tau <: \sigma}{(\tau_1, \ldots, \tau_n)\tau <: (\sigma_1, \ldots, \sigma_n)\sigma} \qquad \frac{P' <: P \quad R <: R'}{P \Rightarrow R <: P' \Rightarrow R'} \qquad \frac{P <: R}{[P] <: [R]}$$

General Expressions

$$\Delta, x : \tau \vdash x : \tau \qquad \frac{\Delta \vdash e : [P] \quad P(p) = \tau}{\Delta \vdash e.p : \tau}$$

$$\Delta \vdash \mathsf{error} : \tau \qquad \Delta \vdash \mathsf{nil} : \tau \qquad \Delta, \ell : \tau \vdash \ell : \tau$$

$$\frac{\Delta \vdash e : R \Rightarrow P \quad \Delta \vdash e_i : \sigma_i \quad \sigma_i <: I_i \quad R = \{r_1 : I_1, \ldots, r_n : I_n\} \quad \Delta \vdash e_j' : \sigma_j'}{\Delta \vdash \mathsf{new}\ e\ \mathsf{with}\ \boldsymbol{e}, \boldsymbol{e'}\ \mathsf{in}\ \boldsymbol{r}, \boldsymbol{r'} : [P]}$$

$$\frac{\Delta \vdash e : I \quad I(f) = (\tau_1)\tau \quad \Delta \vdash e_1 : \sigma_1 \quad \sigma_1 <: \tau_1}{\Delta \vdash e.f(e_1) : \tau}$$

$$\frac{\Delta \vdash e : \tau \quad \Delta(x) = \mathsf{Var}(\sigma) \quad \tau <: \sigma}{\Delta \vdash x := e : \tau} \qquad \frac{\Delta \vdash x : \mathsf{Var}(\sigma)}{\Delta \vdash !x : \sigma}$$

$$\frac{\Delta \vdash e_1 : \tau_1 \quad \Delta, x : \tau_1 \vdash e_2 : \tau_2}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \frac{\emptyset; |\Delta| \vdash_c e : R \Rightarrow P}{\Delta \vdash \mathsf{compose}\ e : R \Rightarrow P}$$

Component Expressions

$$\emptyset; \Delta \vdash_c \boldsymbol{\varepsilon} : \emptyset \Rightarrow \emptyset \qquad \frac{\emptyset; |\Delta| \vdash_c e : R \Rightarrow P}{\emptyset; \Delta \vdash_c \mathsf{compose}\ e : R \Rightarrow P}$$

$$\frac{\mathcal{R}; \Delta, x : I \vdash_c e : R \Rightarrow P}{\mathcal{R}; \Delta \vdash_c \mathsf{require}\ x : I\ \mathsf{and}\ e : R, x : I \Rightarrow P}$$

$$\frac{\mathcal{R}, x : I; \Delta \vdash_c e : R \Rightarrow P}{\mathcal{R}; \Delta \vdash_c \mathsf{provide}\ x : I\ \mathsf{and}\ e : R \Rightarrow P, x : I}$$

$$\frac{|\Delta| \vdash e_1 : \tau \quad \tau <: R' \Rightarrow P' \quad \mathcal{R}, x.R'; x : [P'], \Delta \vdash_c e_2 : R \Rightarrow P}{\mathcal{R}; \Delta \vdash_c \mathsf{intro}\ x : R' \Rightarrow P' = e_1\ \mathsf{and}\ e_2 : R \Rightarrow P}$$

$$\frac{\Delta(\pi_1) <: I \quad \mathcal{R}; \Delta \vdash_c e : R \Rightarrow P}{\mathcal{R}, \pi_2 : I; \Delta \vdash_c \mathsf{plug}\ \pi_1\ \mathsf{into}\ \pi_2\ \mathsf{and}\ e : R \Rightarrow P}$$

$$\frac{\Delta, w : \{m : (\sigma)\tau, \ldots\}, a : \mathsf{Var}(\tau_a), x : \sigma \vdash b : \tau \quad \mathcal{R}; \Delta, w : \{m : (\sigma)\tau, \ldots\} \vdash_c e_2 : R \Rightarrow P}{\mathcal{R}; \Delta \vdash_c \mathsf{meth}\ w : \{a : \tau_a, \ldots; m(x : \sigma) = b, \ldots\}\ \mathsf{and}\ e_2 : R \Rightarrow P}$$

Fig. 9. Type System.

of pending requirements present in the antecedent of typing judgments is a set of type assignments to plugs of either the form $x : I$ (introduced by the rule for provide) or of the form $x.p : I$ (introduced by the rule for intro). Indeed, in the rule for intro the notation $x.R$ stands for the set of plugs $\{x.p_i : I_i \mid R(p_i) = I_i\}$.

This set represents the requirements of the internally introduced component. On the other hand, in the rule for plug, when we write $\mathcal{R}, \pi_2 : I$ we implicitly assume that $\pi_2 : I \notin \mathcal{R}$, that is, the pending requirement is removed when the plugging is performed. Note that subsumption allows a requirement to be satisfied by a service that conforms to some of its subtypes, and the introduction of a more specialized component instead of one of the type mentioned in the intro expression.

3.4 Operational Semantics

Here we present an operational semantics for the component calculus. Our presentation adopts a big step structural semantics and involves again two judgment forms, one applying to general expressions and other to component expressions. Note that we use standard notations for substitutions and store updates, except that when substituting port names by locations in component expressions, we do not descend below compose e subexpressions. Finally, to simplify the specification without loss of generality we consider just a single argument in methods.

The judgment dealing with general expressions has the form $e; S \downarrow v; S'$, where S and S' are heaps, e is a program and v is a result. Such a judgment states that execution of program e in heap S terminates yielding result v and heap S'. As usual, a heap is an assignment of certain values to heap locations $(\ell, \dots)$; we assume given an infinite set of locations, and of an "allocation" function *new* that picks some unused location in a given heap. Every heap associates to each location in its domain either a *heap value*, or another location in its domain. So, when S is a heap, we write $S[\ell \mapsto v]$ for the heap that is identical to S, except that value v is associated to location ℓ. Heap values represent the various entities needed at runtime: we have therefore the heap value nil, which represent uninitialized locations during a link process, *cells*, which represent state variables, *records*, which represent method blocks, and *instances*, which represent (partial) component instances. Note that locations in a heap can hold other locations and define chains of indirection.

A cell holding a value v is represented by $\langle v \rangle$. A record will be represented by a tuple $\{\!| m(x) = b, \dots |\!\}$ of methods with type-erased parameters. An instance will be represented by an *environment*, that is, a set $\{x_1 \mapsto \ell_1, \dots, x_n \mapsto \ell_n\}$ of assignments of locations to names. These names refer to the ports and internal components of the given component instance. It is useful to see an environment as a substitution of locations for names; therefore we will sometimes write $c(e)$ for the result of applying c as a substitution to e.

We will see that possible results of a computation are the null value nil, some location of the final heap, or a component value. A *component value* is a term of component type without free variables or locations, that is, a term such that $\vdash d : P \Rightarrow R$. This fact reflects, in an abstract way, that components are stateless self-contained units open to late composition and instantiation. It is therefore realistic to assume concrete and efficient implementations of the present model, in which components can be manipulated as first-class entities (like instances of

General Expressions

$$\mathsf{compose}\ c; S \downarrow \mathsf{compose}\ c; S \qquad \ell; S \downarrow \ell; S \qquad \frac{S(\ell) = \langle v \rangle}{!\ell; S \downarrow v; S} \qquad \frac{S(\ell) \neq \langle v \rangle}{!\ell; S \downarrow \mathsf{error}; S}$$

$$\frac{e; S \downarrow \ell; S' \quad e_1; S' \downarrow v; S'' \quad S'(\ell) = \{\!|m(x) = b, \dots |\!\} \quad b\{{}^{x}/_{v}\}; S'' \downarrow v'; S'''}{e.m(e_1); S \downarrow v'; S'''}$$

$$\frac{e_1; S \downarrow v_1; S' \quad e_2\{{}^{x}/_{v_1}\}; S' \downarrow v; S''}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2; S \downarrow v; S''} \qquad \frac{e; S \downarrow v; S' \quad v \neq \mathsf{nil} \quad S'(v) \neq \{\!|m(x) = b, \dots |\!\}}{e.m(e_1); S \downarrow \mathsf{error}; S'}$$

$$\mathsf{nil}; S \downarrow \mathsf{nil}; S \qquad \frac{e; S \downarrow d; S' \quad \emptyset; d, S' \Downarrow s; S^1 \quad e_i; S^i \downarrow v_i; S^{i+1} \quad v_i \neq \mathsf{nil}}{\mathsf{new}\ e\ \mathsf{with}\ e_n\ \mathsf{in}\ r_n; S \downarrow \ell; S^{n+1}[\ell \mapsto s][s(r_i) \mapsto v_i]}$$

$$\frac{e; S \downarrow v; S'}{x := e; S \downarrow v; S'[x \mapsto \langle v \rangle]} \qquad \frac{e; S \downarrow \ell; S' \quad S'(\ell) = \{p \mapsto \ell', \dots\} \quad v = \mathit{deref}_{S'}(\ell')}{e.p; S \downarrow v; S'}$$

$$\frac{e; S \downarrow \ell; S' \quad \ell \neq \mathsf{nil} \quad S'(\ell) \neq \{p \mapsto \ell', \dots\}}{e.p; S \downarrow \mathsf{error}; S'}$$

$$\frac{e; S \downarrow \ell; S' \quad S'(\ell) = \{p \mapsto \ell', \dots\} \quad \mathit{deref}_{S'}(\ell') = \mathsf{nil}}{e.p; S \downarrow \mathsf{error}; S'}$$

Component Expressions

$$c; \boldsymbol{\varepsilon}; S \Downarrow c; S \qquad \frac{c; e; S \Downarrow c'; S'}{c; \mathsf{compose}\ e; S \Downarrow c'; S'}$$

$$\frac{c\{x \mapsto \ell\}; e\{{}^{x}/_{\ell}\}; S[\ell \mapsto \mathsf{nil}] \Downarrow c'; S'}{c; \mathsf{require}\ x : I\ \mathsf{and}\ e; S \Downarrow c'; S'}\ \dagger \qquad \frac{c\{x \mapsto \ell\}; e\{{}^{x}/_{\ell}\}; S[\ell \mapsto \mathsf{nil}] \Downarrow c'; S'}{c; \mathsf{provide}\ x : I\ \mathsf{and}\ e; S \Downarrow c'; S'}\ \dagger$$

$$\frac{\mathsf{new}\ e_1; S \downarrow c'; S' \quad c\{x \mapsto c'\}; e_2\{{}^{x}/_{c'}\}; S' \Downarrow c''; S'}{c; \mathsf{intro}\ x : e_1\ \mathsf{and}\ e_2; S \Downarrow c''; S'}$$

$$\frac{c\{w \mapsto \ell\}; e\{{}^{w}/_{\ell}\}; S[\ell_a \mapsto \langle \mathsf{nil} \rangle][\ell \mapsto \{\!|m(x : \sigma) = c(b)\{{}^{a}/_{\ell_a}\}\{{}^{w}/_{\ell}\}, \dots |\!\}] \Downarrow c'; S'}{c; \mathsf{meth}\ w : \{a : \tau_a, \dots ; m(x : \sigma) = b, \dots\}\ \mathsf{and}\ e; S \Downarrow c'; S'}\ \ddagger$$

$$\frac{\mathrm{select}_S(\pi_2) = \ell \quad c; e; S[\ell \mapsto \mathrm{select}_S(\pi_1)] \Downarrow c'; S'}{c; \mathsf{plug}\ \pi_1\ \mathsf{into}\ \pi_2\ \mathsf{and}\ e; S \Downarrow c'; S'}$$

$$\frac{c; e; S \Downarrow c'; S' \quad \neg\mathrm{select}_S(\pi_2)}{c; \mathsf{plug}\ \pi_1\ \mathsf{into}\ \pi_2\ \mathsf{and}\ e; S \Downarrow c'; S'}$$

$$\frac{\neg\mathrm{select}_S(\pi_1)}{c; \mathsf{plug}\ \pi_1\ \mathsf{into}\ \pi_2\ \mathsf{and}\ e; S \Downarrow \mathsf{error}; S'}$$

†) $\ell = \mathit{new}(S)$.
‡) $\ell, \ell_a = \mathit{new}(S)$.

Fig. 10. Operational semantics.

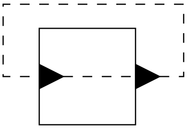


Fig. 11. Cyclic dependency.

class `Class` in Java, which can be dynamically loaded and instantiated [18]) and possibly transmitted across communication channels.

Derivable judgments are defined by the system of rules given in Fig.10. The evaluation relation applies to valid configurations which are represented by a heap-expression pairs $e; S$ such that e is a program without free variables, and that all locations mentioned in e belong to the domain of S. The specification of $\downarrow$ relies on an auxiliary relation $c; e; S \Downarrow c'; S'$ that defines the dynamic component assembly process, and is used in the rule for new. The intended meaning of $\Downarrow$ can be explained as follows: c is a given, partially assembled, component instance; e is an expression of component type specifying further assembly operations and S an initial heap. Then, whenever $c; e; S \Downarrow c'; S'$ is derivable, c' is the result of completing c as stated by e, and S' the final heap w.r.t. c' must be understood (ie. locations in c' are interpreted in S'). On the other hand, the specification of $\Downarrow$ also relies on $\downarrow$ in the rule for intro. Before discussing the rules specifying $\downarrow$ we address the rules for $\Downarrow$. The rules for ε and compose are clear. The rules for require and provide are similar; their effect is just to add an additional binding to the current instance c: the contents of the (new) location bound to the port name is left undefined in the updated heap, and will be later assigned by some operation of plugging or instantiation. Note that, in each case, the port name x is substituted by the given location throughout the continuation e. The rule for intro recursively assembles an instance c' according to e_1 and updates the current instance c and heap as expected.

The rule for meth introduces a new location for each local state variable of the method's block and a new record. The state variable names are substituted for the corresponding locations in the body of the methods. Note that the updated instance $c\{w \mapsto \ell\}$ is used as a substitution, and applied to the method bodies, so to replace all references to the ports and inner components of the instance under assembly by the locations previously assigned to them. Therefore, method bodies in records do not contain occurrences of free variables other than their parameters.

The rules for plug can now be easily understood: plugging a source port π_1 to a destination port π_2 amounts to storing the location assigned to π_1 into the location assigned to π_2, if the destination port actually exists in the target component — something that may not happen due to sub-typing of component types. This situation is handled by the last rule for plug, which does nothing. Note the use of function $\text{select}_S(-)$ to extract the location associated to a plug, which is defined by

$$\text{select}_S(x) := S(x) \quad \text{select}_S(x.p) := \text{select}_{S(x)}(p)$$

When such location does not exist, we write $\neg\text{select}_\pi(S)$.

This encoding of plugs introduces chains of indirection: such chains are transversed when component instance ports are selected. If no vacuous cyclic dependencies among components are introduced at assembly time — as in Fig.11 — then all such chains of indirection are finite. Note that a vacuous cyclic dependencies can cause a port selection operation to get stuck instead of the link process that creates the cycle during instantiation. Therefore, the previous discussion explains the use of the auxiliary function *deref* in the rule for port selection $e.p$; $deref_S(\ell)$ denotes the last location in the chain starting at ℓ that refers to a heap value, whenever such chain terminates. The remaining rules for $\downarrow$ deserve no special comment; except perhaps the one for new; therein the values of the expressions e_i are plugged into the matching requirements of the newly created instance s.

The remaining rules generate the error value in the appropriate circumstances, which are: invocation of inexistent method, selection of inexistent or null component port and plug with inexistent source port. Note that we have omitted the usual rules for error propagation.

3.5 Type Safety

Our aim now is to state a type safety result for the core calculus of components (see [22] for the full proofs).

We start by defining a notion of typing for heaps. For technical reasons, we introduce a new kind of types, so to type partially linked component instances. Such generalized instance types will have the form $[R \Rightarrow P]$ whenever $R \Rightarrow P$ is a component type; $[\emptyset \Rightarrow P]$ will be identified with the instance type $[P]$. We first need to introduce a few auxiliary concepts.

Definition 1. (Locations and references)

1. (Direct location) *A location* ℓ *is* direct *when* $S(\ell) \neq \mathsf{nil}$ *and* $S(\ell) \notin \mathsf{Loc}$.
2. (Leads to) *When* $\mathrm{deref}_S(\ell) = \ell'$ *we say that* ℓ leads to ℓ' *is* S.
3. (Refers to) *A location* ℓ refers to *a heap value* v *in* S *when* ℓ *leads to* ℓ' *in* S *and* $S(\ell') = v$.
4. (Undefined location) *A location is undefined when it refers to no value. In other words, it starts a cyclic chain of references.*

Definition 2. (Typing of Heaps) *A typing context* Γ types *heap* S *if, for all* ℓ *in* S, $\Gamma(\ell) = \tau$ *implies that* ℓ *is either undefined or refers to a heap value of a type* $\tau' <: \tau$ *w.r.t.* S *and* Γ.

v is a heap value of type τ w.r.t. S and Γ whenever $v = \mathsf{nil}$ or

— $\tau = \mathsf{Var}(\sigma)$ *and* $v = \langle u \rangle$ *with either* $u = \mathsf{nil}$, *u a component value of a subtype of σ, or a direct location such that* $\Gamma(u) <: \sigma$, *or*

— $\tau = \{m_1 : (\boldsymbol{\sigma}_1)\beta_1, \dots\}$ *and* $v = \{\!|m_1(\boldsymbol{x}_1) = b_1, \dots|\!\}$, *with* $\Gamma, \boldsymbol{x}_i : \boldsymbol{\sigma}_i \vdash b_i : \beta_i$ *for all* m_i, *or*

— $\tau = [\boldsymbol{r} : \boldsymbol{I} \Rightarrow \boldsymbol{p} : \boldsymbol{J}]$ *and* $v = \{\boldsymbol{r} \mapsto \boldsymbol{\ell^r}, \boldsymbol{p} \mapsto \boldsymbol{\ell^p}, \dots\}$, *such that each* ℓ_i^r *is either undefined or refers to a heap value of a subtype of* I_i *w.r.t.* S *and* Γ,

and each $\boldsymbol{\ell^p}$ is either undefined, leads to some ℓ_i^r, or refers to a record value of a subtype of J_i w.r.t. S and Γ.

Remember that a location can be undefined just because of cyclic reference chains. We can now state

Proposition 1. (Subject Reduction) *Let $e; S$ be a valid configuration and Γ type S. If $\Gamma \vdash e : \tau$ and $e; S \downarrow v; S'$ then*
a) there is Γ' extending Γ and typing S'.
b) $\Gamma' \vdash v : \tau'$ for some $\tau' <: \tau$.
c) v is either nil*, a direct location in S' or a component value.*

This proposition shows that well-typed programs do not get stuck due to invalid, undefined method calls, or illegal component assembly operations because the resulting value of a computation is never error. However, programs can get stuck due to cyclic dependencies or nil references (caused by uninitialized state variables).

4 Some Implementation Issues

In this section, we discuss the current status of an experimental implementation of ComponentJ. A preprocessor performs type-checking of ComponentJ source files and generates standard Java files as output. The implemented type system adopts name equivalence of interface types, which is directly supported by the JVM, instead of structural equivalence as used in the core calculus.

A ComponentJ source file can define a component, a component type (similar to a Java interface) or an interface type (exactly like Java interfaces), as illustrated in Sec.2. Inside each component, besides method blocks, pluggings, inner components and so on, ComponentJ also allows the declaration and use of standard Java classes, for strict local use of the component, although such classes are not allowed to extend classes defined outside the component in order to enforce the black-box approach. Each method block is also represented by a local class. Plugging between ports is performed by propagation of references, by a process close to the one used in the operational semantics for the core calculus, but direct references to method blocks are cached at each port during component initialization. Consequently, any provided service port conforming to some interface type directly references a Java object implementing such interface, unlike in usual nested structures of components, where some overhead often occurs in explicit method forwarding throughout layers.

ComponentJ generates for each component definition file a Java source file for a class adhering to a meta-level protocol for use of ComponentJ, and aimed at supporting interoperatibiliy of ComponentJ generated classes and independently developed components. Such protocol is still under development, and will define how component types are represented by Java types, and how the dynamic linking process is to be managed.

Concerning initialization, an issue which is not addressed in the core calculus, the definition of a "constructor" is allowed in each component. The body of such a constructor consists of Java code in which any service provided by inner components or imported through a required interface may be called. Such constructor is called at the end of the instantiation and link phases that occur at component instantiation time.

5 Related Work

Our proposal is strongly related to recent work on first-class modules with external linking instructions [15,7,12,9,1], most prominently with the UNITS of MzScheme reported in [9], and the primitive modules of [1]. However, UNITS are more close to conventional modules in standard imperative and functional programs (even with parametric modules like in SML [13]), linked to yield closed programs, than to instantiable components offering object-like interfaces. A more recent work [8] demonstrates the use of UNITS in an object-oriented context, but again units are used as modules declaring classes. Such modules can then be used to structure class frameworks rather than to compose individual components through aggregation. On the other hand, the approach of [1] does not allow manipulation of components as first-class entities in the core language, something which is considered an important aspect of COP. Another distinguishing feature of our model is that all dependences of a component on its environment must be brought explicit in its interfaces, unlike in [9,1] where modules can be open (contain free variables).

6 Conclusions and Future Work

In this paper, we studied some statically type-checkable programming language constructs intended for supporting component-oriented programming styles. The model was motivated and presented by a core imperative typed formal calculus, whose operational semantics is shown to enjoy a type-safety property. We also discussed a still evolving implementation of a compiler for ComponentJ, an extension to the Java language theoretically backed by the proposed core component calculus. Type-safe separate compilation of components allows the late binding and instantiation of components of subtype of the types statically checked. We envisage two parallel directions of future work. On the one hand, we would like to extend the core calculus and type system to handle more sophisticated types useful for COP, for instance parametric, bounded, dynamic and container types. For example, we are currently developing an extension of the type system proposed here which allowed for the type of output ports to be defined from the static types of the ports plugged into its requirements, a feature that adds significant expressiveness to the calculus. On the other hand, we are currently engaged into studying improved implementation techniques for supporting the component framework and its interoperatibility with other component models.

Acknowledgements. We thank the anonymous referees for their helpful comments.

References

1. Davide Ancona and Elena Zucca. A primitive calculus of module systems. In Michael Hanus and Mario Rodríguez-Artalejo, editors, *Algebraic and Logic Programming, 5th International Conference, ALP'96*, volume 1139 of *Lecture Notes in Computer Science*, pages 179–193, Aachen, Germany, 25–27 September 1996. Springer-Verlag.
2. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In Rachid Guerraoui, editor, *ECOOP '99 — Object-Oriented Programming 13th European Conference, Lisbon Portugal*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer-Verlag, New York, N.Y., June 1999.
3. Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* Ph.D. thesis, Dept. of Computer Science, University of Utah, March 1992.
4. Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, October 1990. ACM Press.
5. Martin Buchi and Wolfgang Weck. Compound types for Java. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 362–373, New York, 1998. ACM Press.
6. Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, N.Y., 1991.
7. Luca Cardelli. Program fragments, linking, and modularization. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, France, 15–17 January 1997.
8. Robert Bruce Findler and Matthew Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 94–104. ACM, June 1999.
9. Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 236–248, Montreal, Canada, 17–19 June 1998.
10. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 171–183, New York, N.Y., January 1998. ACM.
11. I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Ira Forman, "Release-to-Release Binary Compatibility in SOM". *ACM SIGPLAN Notices OOPSLA'95*, 30(10):426–438, October 1995.
12. Neal Glew and Greg Morrisett. Type-safe linking and modular assembly language. In *Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 250–261, New York, N.Y., January 1999. ACM.

13. Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Portland, Oregon, January 17–21, 1994*, pages 123–137, New York, NY, USA, 1994. ACM Press.
14. JavaSoft. JavaBeans^{TM}. http://java.sun.com/beans, December 1996. Version 1.00-A.
15. Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In ACM, editor, *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Portland, Oregon, January 17–21, 1994*, pages 138–150, New York, NY, USA, 1994. ACM Press.
16. Dinesh Katiyar, David Luckham, John Mitchell, and Sigurd Meldal. Polymorphism and subtyping in interfaces. *ACM SIGPLAN Notices*, 29(8):22–34, August 1994.
17. John Lamping. Typing the specialization interface. In Andreas Paepcke, editor, *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 201–214, Washington, DC, USA, 1993.
18. Sheng Liang and Gilad Bracha. Dynamic class loading in the Java$^{\text{TM}}$ virtual machine. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 36–44. ACM Press, 1998.
19. Boris Magnusson. Code reuse considered harmful. *J. of Object-Oriented Programming*, 4(3), November 1991.
20. Thomas J. Mowbray and William A. Ruh. *Inside CORBA: Distributed Object Standards and Applications.* Addison-Wesley, Reading, MA, USA, 1997.
21. Dale Rogerson. *Inside COM: Microsoft's Component Object Model.* Microsoft Press, 1997.
22. João Costa Seco and Luís Caires. A basic model of typed components. Technical report, Departamento de Informática, Universidade Nova de Lisboa, 2000.
23. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, N.Y., 1998.
24. David Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3):223–242, July 1991.
25. Wolfgang Weck. Inheritance using contracts and object composition. In Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors, *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, pages 105–12. Turku Centre for Computer Science, September 1997.
26. Wolfgang Weck and Clemens Szypersky. Do we need inheritance? In *Proceedings of the Workshop on Composability Issues on Object-Orientation (ECOOP'96)*, 1996.

On Inner Classes

Atsushi Igarashi[1]* and Benjamin C. Pierce[2]

[1] Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan
igarashi@is.s.u-tokyo.ac.jp
[2] Department of Computer & Information Science, University of Pennsylvania
200 South 33rd St., Philadelphia, PA 19104, USA
bcpierce@cis.upenn.edu

Abstract. Inner classes in object-oriented languages play a role similar to nested function definitions in functional languages, allowing an object to export other objects with direct access to its own methods and instance variables. However, the similarity is deceptive: a close look at inner classes reveals significant subtleties arising from their interactions with inheritance.

The goal of this work is a precise understanding of the essential features of inner classes; our object of study is a fragment of Java with inner classes and inheritance (and almost nothing else). We begin by giving a *direct* reduction semantics for this language. We then give an alternative semantics by *translation* into a yet smaller language with only top-level classes, closely following Java's Inner Classes Specification. We prove that the two semantics coincide, in the sense that translation commutes with reduction, and that both are type-safe.

1 Introduction

It has often been observed that the gap between object-oriented and functional programming styles is not as large as it might first appear; in essence, an object is just a record of function closures. However, there are differences as well as similarities. On the one hand, objects and classes incorporate important mechanisms not present in functions (static members, inheritance, object identity, access protection, etc.). On the other hand, functional languages usually allow *nested* definitions of functions, giving inner functions direct access to the local variables of their enclosing definitions.

A few object-oriented languages do support this sort of nesting. For example, Smalltalk [8] has special syntax for "block" objects, similar to anonymous functions. Beta [15] provides patterns, unifying classes and functions, that can be nested arbitrarily. More recently, *inner classes* have been popularized by their inclusion in Java 1.1 [9, 12].

Inner classes are useful when an object needs to send another object a chunk of code that can manipulate the first object's methods and/or instance variables.

* This work was done while the author was visiting University of Pennsylvania.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 129–153, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Such situations are typical in user-interface programming: for example, Java's Abstract Window Toolkit [4] allows a *listener object* to be registered with a user-interface component such as a button; when the button is pressed, the `actionPerformed` method of the listener is invoked. For example, suppose we want to increment a counter when a button is pressed. We begin by defining a class `Counter` with an inner class `Listener`:

```
class Counter {
  int x;
  class Listener implements ActionListner {
    public void actionPerformed(ActionEvent e) { x++; }
  }
  void listenTo(Button b) {
    b.addActionListener(new Listener());
  }
}
```

In the definition of the method `actionPerformed`, the field `x` of the enclosing `Counter` object is changed. The method `listenTo` creates a new listener object and sends it to the given `Button`. Now we can write

```
Counter c = new Counter();
Button b = new Button("Increment");
c.listenTo(b);
gui.add(b);
```

to create and display a button that increments a counter every time it is pressed.[1]

Inner classes are a powerful abstraction mechanism, allowing programs like the one above to be expressed much more conveniently and transparently than would be possible using only top-level classes. However, this power comes at a significant cost in complexity: inner classes interact with other features of object-oriented programming—especially inheritance—in some quite subtle ways. For example, a closure in a functional language has a simple lexical environment, including all the bindings in whose scope it appears. An inner class, on the other hand, has access, via methods inherited from superclasses, to a *chain* of environments—including not only the lexical environment in which it appears, but also the lexical environment of each superclass. Conversely, the presence of inner classes complicates our intuitions about inheritance. What should it mean, for example, for an inner class to inherit from its enclosing class? What happens if a top-level class inherits from an inner class defined in a different top-level class?

JavaSoft's Inner Classes Specification [12] provides one answer to these questions by showing how to translate a program with inner classes into one using only top-level classes, adding to each inner class an extra field that points to an instance of the enclosing class. This specification gives clear basic intuitions

[1] Strictly speaking, the increment of `x` should be `synchronized` with the listener's own counter, written `Counter.this`: listener methods are generally triggered in a thread different from the constructor thread of the current object.

about the behavior of inner classes, but it falls short of a completely satisfying account. First, the style is indirect: it forces programmers to reason about their code by first passing it through a rather heavy transformation. Second, the document itself is somewhat imprecise, consisting only of examples and English prose. Different compilers (even different versions of Sun's JDK) have interpreted the specification differently in some significant ways (cf. Section 6).

The goal of this work is a precise understanding of the essential features of inner classes. Our main contributions are threefold:

– First, we give a direct operational semantics and typing rules for a small language with inner classes and inheritance. The typing rules are shown to be sound for the operational semantics in the standard sense. To our knowledge, this direct style of semantics is formalized for the first time.
  To keep the model as simple as possible, we focus on the most basic form of inner classes in Java, omitting the related mechanisms of anonymous classes, local classes within blocks, and static nested classes. Also, we do not deal with the (important) interactions between access annotations (`public`/`private`/ etc.) and inner classes (cf. [12, 2, 1]).
– Next, we give a translation from the language with inner classes to an even smaller language with only top-level classes, formalizing the translation semantics of the Java Inner Classes Specification. We show that the translation preserves typing.
– Finally, we prove that the two semantics define the same behavior for inner classes, in the sense that the translation commutes with the high-level reduction relation in the direct semantics. This property, together with the property of preservation of typing, guarantees correctness of the translation semantics with respect to the direct semantics, for the case where whole programs are being translated.
  The case where some translated classes are linked with classes written directly in the target language is more subtle, and we do not handle it here. The main desired theorem in this case would be *full abstraction*, which states that translated expressions that can be distinguished by a target language context can also be distinguished in the source language. Unfortunately, our translation is not fully abstract, because our modeling language does not include private fields, which are used by the real translation to prevent observers from directly accessing the field of an inner class instance that holds a pointer to its containing object. (The question of full abstraction for full-scale inner class translations has been considered by Abadi [1] and Pugh [2].)

Recently, Glew [7] has studied closure conversion in the context of an object calculus without classes; our translation semantics can be viewed as closure conversion of class definitions. However, since his calculus does not have classes, a semantic account of interaction between inheritance and nested classes has not been given.

The basis of our work is a core calculus called Featherweight Java, or FJ. This calculus was originally proposed in the context of a formal study [10] of GJ [3], an extension of Java with parameterized classes. It was designed to omit as many features of Java as possible (even assignment), while maintaining the essential flavor of the language and its type system. Its definition fits comfortably on a page, and its basic properties can be proved with no more difficulty than, say, those of the simply typed lambda-calculus with subtyping. This extreme simplicity makes it an ideal vehicle for the rigorous study of new language features such as parameterized classes and inner classes.

The remainder of the paper is organized as follows. Section 2 briefly reviews Featherweight Java. Section 3 defines FJI, an extension of FJ with inner classes, giving its syntax, typing rules, and reduction rules, and stating standard type soundness results. Section 4 defines a compilation from FJI to FJ, modeling the translation semantics of the Inner Classes Specification, and proves its correctness with respect to the direct semantics in the previous section. Section 5 discusses the elaboration process from user programs to FJI, which is considered an intermediate language to define semantics. Section 6 examines some behavioral differences between compilers resulting from inconsistencies in the existing specification, Section 7 discusses related work, and Section 8 offers concluding remarks.

For brevity, proofs of theorems are omitted; they appear in a companion technical report [11].

2 Featherweight Java

We begin by reviewing the basic definitions of Featherweight Java [10]. FJ is a tiny fragment of Java, including only top-level class definitions, object instantiation, field access, and method invocation. (The original version of FJ also included typecasts, which are required to model the compilation of GJ into Java. They are omitted from this paper, since they do not interact with inner classes in any significant way.) Our main goal in designing FJ was to make a proof of type soundness ("well-typed programs don't get stuck") as concise as possible, while still capturing the essence of the soundness argument for the full Java language. Any language feature that made the soundness proof *longer* without making it significantly *different* was a candidate for omission. Even assignments are omitted from FJ, as well as advanced features such as reflection and concurrency. Since FJ is a sublanguage of the extension defined in Section 3, we just show its syntax and an example of program execution here. The rest of the definition can be found in Figure 4.

The abstract syntax of FJ class declarations, constructor declarations, method declarations, and expressions is given as follows:

$$\begin{aligned}
\texttt{L} &::= \texttt{class C extends C \{}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{M}}\texttt{\}}\\
\texttt{K} &::= \texttt{C(}\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{) \{super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{;\}}\\
\texttt{M} &::= \texttt{C m(}\overline{\texttt{C}}\ \overline{\texttt{x}}\texttt{) \{return e;\}}\\
\texttt{e} &::= \texttt{x} \mid \texttt{e.f} \mid \texttt{e.m(}\overline{\texttt{e}}\texttt{)} \mid \texttt{new C(}\overline{\texttt{e}}\texttt{)}
\end{aligned}$$

The metavariables A, B, C, D, and E range over class names; f and g range over field names; m ranges over method names; x ranges over parameter names; `c`, `d` and `e` range over expressions; `L` ranges over class declarations; `K` ranges over constructor declarations; and `M` ranges over method declarations. We write $\overline{\mathtt{f}}$ as shorthand for $\mathtt{f}_1,\ldots,\mathtt{f}_n$ (and similarly for $\overline{\mathtt{C}}$, $\overline{\mathtt{x}}$, $\overline{\mathtt{e}}$, etc.) and write $\overline{\mathtt{M}}$ as shorthand for $\mathtt{M}_1\ldots\mathtt{M}_n$ (with no commas). We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. The length of a sequence $\overline{\mathtt{x}}$ is written $\#(\overline{\mathtt{x}})$. We abbreviate operations on pairs of sequences in the obvious way, writing "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$" as shorthand for "$\mathtt{C}_1\ \mathtt{f}_1,\ldots,\mathtt{C}_n\ \mathtt{f}_n$" and "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$;" as shorthand for "$\mathtt{C}_1\ \mathtt{f}_1;\ldots\mathtt{C}_n\ \mathtt{f}_n;$" and "$\mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;}$" as shorthand for "$\mathtt{this.f}_1\mathtt{=f}_1\mathtt{;}\ldots\mathtt{this.f}_n\mathtt{=f}_n\mathtt{;}$". For the sake of conciseness, we often abbreviate the keyword `extends` to the symbol `extends` and the keyword `return` to the symbol `return`. Sequences of field declarations, parameter names, and method declarations are assumed to contain no duplicate names.

A key simplification in FJ is the omission of assignment, making FJ purely functional. It is realized by assuming that all fields and method parameters are implicitly marked `final`. (Of course, most *useful* examples of programming in Java do involve its side-effecting features, and inner classes do interact with assignment: in particular, if inner classes may appear inside method definitions, then local variables of the enclosing method must be marked `final` if they are mentioned in an inner class. To handle this feature, our model would need to be extended with assignment. However, we do not need it for the present modeling task, and, by omitting assignment from FJ and FJI, we obtain a much simpler model that offers just as much insight into inner classes.) An object's fields are initialized by its constructor and never changed afterwards. Moreover, a constructor has a stylized syntax such that there is one parameter for each field, with the same name as the field; the `super` constructor is invoked on the fields of the supertype; and the remaining fields are initialized to the corresponding parameters. (These constraints are enforced by the typing rules.) This stylized syntax makes the operational semantics simple: a field access expression $\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).f}_i$ just reduces to the corresponding constructor argument $\mathtt{e}_i$. Also, since FJ does not have assignment statements, a method body always consists of a single `return` statement: all the computation in the language goes on in the expressions following these `return`s. A method invocation expression $\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)}$ is reduced by looking up the expression $\mathtt{e}_0$ following the `return` of method `m` in class `C` in the class table, and reducing to the instance of $\mathtt{e}_0$ in which $\overline{\mathtt{d}}$ and the receiver object ($\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)}$) are substituted for formal arguments and the special variable `this`, respectively. Figure 4 states these reduction rules precisely.

A program in FJ is a pair of a *class table* (a set of class definitions) and an expression (corresponding to the `main` method in a Java program). The reduction relation is of the form $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression `e` reduces to expression $\mathtt{e}'$ in one step."

For example, given the class definitions

```
class A extends Object {
  A() { super(); }
}
```

```
class B extends Object {
  B() { super(); }
}
class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd;
  }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
}
```

the expression `new Pair(new A(), new B()).setfst(new B())` reduces to `new Pair(new B(), new B())` as follows

$$\begin{array}{ll} & \underline{\texttt{new Pair(new A(), new B()).setfst(new B())}} \\ \longrightarrow & \texttt{new Pair(new B(), }\underline{\texttt{new Pair(new A(), new B()).snd}}\texttt{)} \\ \longrightarrow & \texttt{new Pair(new B(), new B())} \end{array}$$

where the underlined subexpressions are the ones being reduced at each step.

3 FJ with Inner Classes

We now define the language FJI by extending FJ with inner classes. Like FJ, FJI imposes some syntactic restrictions to simplify its operational semantics: (1) receivers of field access, method invocation, or inner class constructor invocation must be explicitly specified (no implicit `this`); (2) type names are always absolute paths to the classes they denote (no short abbreviations); and (3) an inner class instantiation expression $\texttt{e}_0\texttt{.new C(}\overline{\texttt{e}}\texttt{)}$ is annotated with the static type `T` of $\texttt{e}_0$, written $\texttt{e}_0\texttt{.new<T> C(}\overline{\texttt{e}}\texttt{)}$.

Because of conditions (2) and (3), FJI is not quite a subset of Java (whereas FJ is); instead, we view FJI as an intermediate language, to which the user's programs are translated by a process of *elaboration*. We describe the elaboration process only informally in this paper (in Section 5), since it is rather complex but not especially deep, consisting mainly of a large number of rules for abbreviating long qualified names; a detailed treatment is given in the companion technical report [11]. We begin with a brief discussion of the key idea of *enclosing instances*.

3.1 Enclosing Instances

Consider the following FJI class declaration:

```
class Outer extends Object {
  Pair p;
  Outer(Pair p) {super(); this.p = p;}
  class Inner extends Object {
    Inner() {super();}
    Object snd_p { return Outer.this.p.snd; }
  }
  Outer.Inner make_inner () { return this.new<Outer> Inner(); }
}
```

Conceptually, each instance `o` of the class `Outer` contains a specialized version of the `Inner` class, which, when instantiated, yields instances of `Outer.Inner` that refer to `o`'s instance variable `p`. The object `o` is called the *enclosing instance* of these `Outer.Inner` objects.

This enclosing instance can be named explicitly by a "qualified `this`" expression (found in both Java and FJI), consisting of the simple name of the enclosing class followed by "`.this`". In general, the class $\mathtt{C}_1.\cdots.\mathtt{C}_n$ can refer to $n-1$ enclosing instances, $\mathtt{C}_1$`.this` to $\mathtt{C}_{n-1}$`.this`, as well as the usual `this`, which can also be written $\mathtt{C}_n$`.this`. To avoid ambiguity of the meaning of `C.this`, the name of an inner class must be different from any of its superclasses.

In FJI, an object of an inner class is instantiated by an expression of the form $\mathtt{e}_0$`.new<T> C(`$\bar{\mathtt{e}}$`)`, where $\mathtt{e}_0$ is the enclosing instance and `T` is the static type of $\mathtt{e}_0$. The result of $\mathtt{e}_0$`.new<T> C(`$\bar{\mathtt{e}}$`)` is always an instance of `T.C`, regardless of the run-time type of $\mathtt{e}_0$. (We avoid a notation like $\mathtt{e}_0$`.new T.C(`$\bar{\mathtt{e}}$`)` because it is *not* in the Java syntax. Java allows only the notation `new T.C(`$\bar{\mathtt{e}}$`)` (without a prefix), which roughly means an instantiation from the class `T.C` with an enclosing instance `T.this`; see Section 5 for more details.) This rigidity reflects the static nature of Java's translation semantics for inner classes. The explicit annotation `<T>` is used in FJI to "remember" the static type of $\mathtt{e}_0$. (By contrast, inner classes in Beta are *virtual* [14], i.e., different constructors may be invoked depending on the run-time type of the enclosing instance; for example, if there were a subclass `Outer`′ of the class `Outer` that also had an inner class `Inner`, then `o.new Inner()` might build an instance of either `Outer.Inner` or `Outer`′`.Inner`, depending on the dynamic type of `o`.)

The elaboration process allows type names to be abbreviated in Java programs. For example, the FJI program above can be written

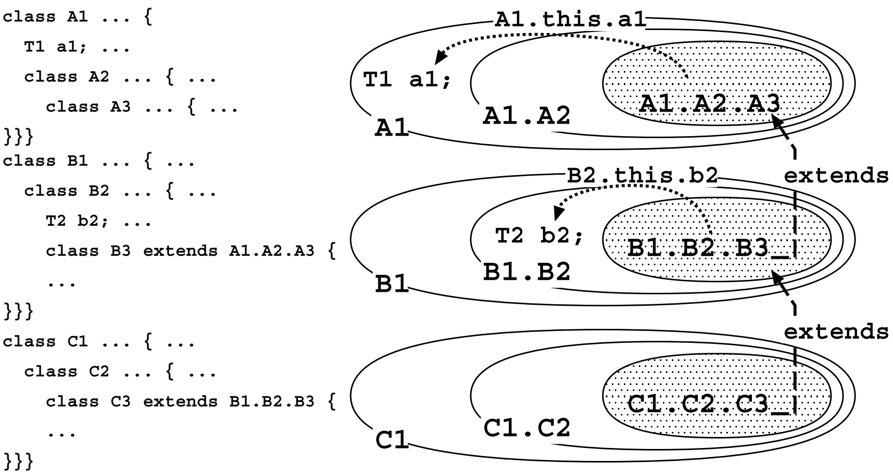
```
class Outer extends Object {
  Pair p;
  Outer(Pair p) {super(); this.p = p;}
  class Inner extends Object {
    Inner() {super();}
    Object snd_p () { return p.snd; }
  }
  Inner make_inner () { return new Inner(); }
}
```

in Java. Here, the return type `Inner` of the `make_inner` method denotes the nearest `Inner` declaration. Also, in Java, enclosing instances can be omitted when they are `this` or a qualified `this`. Thus, `this.new<Outer> Inner()` from the original example is written `new Inner()` here.

3.2 Subclassing and Inner Classes

Almost any form of inheritance involving inner classes is allowed in Java: a top-level class can extend an inner class of another top-level class, or an inner class can extend another inner class from a completely different top-level class. An inner class can even extend its own enclosing class. (Only one case is disallowed: a class cannot extend its own inner class. We discuss the restriction later.) This liberality, however, introduces significant complexity because a method inherited from a superclass must be executed in a "lexical environment" different from the subclass's. Figure 1 shows a situation where three inner classes, `A1.A2.A3` and `B1.B2.B3` and `C1.C2.C3`, are in a subclass hierarchy. Each white oval represents an enclosing instance and the three shaded ovals indicate the regions of the program where the methods of a `C1.C2.C3` object may have been defined. A method inherited from `A1.A2.A3` is executed under the environment consisting of enclosing instances `A1.this` and `A2.this` and may access members of enclosing classes via `A1.this` and `A2.this`; similarly for `B1.B2.B3` and `C1.C2.C3`. In general, when a class has n superclasses which are inner, n different environments may be accessed by its methods. Moreover, each environment may consist of more than one enclosing instance; six enclosing instances are required for all the methods of `C1.C2.C3` to work in the example above.

From the foregoing, we see that we will have to provide, in some way, six enclosing instances to instantiate a `C1.C2.C3` object. Recall that, when an object



```
class A1 ... {
  T1 a1; ...
  class A2 ... { ...
    class A3 ... { ...
}}}
class B1 ... { ...
  class B2 ... { ...
    T2 b2; ...
    class B3 extends A1.A2.A3 {
    ...
}}}
class C1 ... { ...
  class C2 ... { ...
    class C3 extends B1.B2.B3 {
    ...
}}}
```

Fig. 1. A chain of environments

of an inner class is instantiated, the enclosing object is provided by a prefix $\mathtt{e}_0$ of the new expression. For example, a C1.C2.C3 object is instantiated by writing $\mathtt{e}_0$.new<C1.C2> C3($\overline{\mathtt{e}}$), where $\mathtt{e}_0$ is the enclosing instance corresponding to C2.this. Where do the other enclosing instances come from?

First, enclosing instances from enclosing classes other than the immediately enclosing class, such as C1.this, do not have to be supplied to a new expression explicitly, because they can be reached via the direct enclosing instance—for example, the enclosing instance $\mathtt{e}_0$ in $\mathtt{e}_0$.new<C1.C2> C3($\overline{\mathtt{e}}$) has the form new C1($\overline{\mathtt{c}}$).new<C1> C2($\overline{\mathtt{d}}$), which includes the enclosing instance new C1($\overline{\mathtt{c}}$) that corresponds to C1.this.

Second, the enclosing instances of superclasses are determined by the constructor of a subclass. Taking a simple example, suppose we extend the inner class Outer.Inner. An enclosing instance corresponding to Outer.this is required to make an instance of the subclass. Here is an example of a subclass of Outer.Inner:

```
class RefinedInner extends Outer.Inner {
  Object c;
  RefinedInner(Outer this$Outer$Inner, Object c) {
    this$Outer$Inner.super(); this.c=c;
}}
```

In the declaration of the RefinedInner constructor, the ordinary argument this$Outer$Inner becomes the enclosing instance prefix for the super constructor invocation, providing the value of Outer.this referred to in the inherited method snd_p. Similarly, in the C1.C2.C3 example, the subclass B1.B2.B3 is written as follows (we assume A1.A2.A3 has a field a3 of type Object):

```
class B1 extends ... { ...
  class B2 extends ... { ...
    class B3 extends A1.A2.A3 {
      Object b3;
      B3(Object a3, A1.A2 this$A1$A2$A3, Object b3) {
        this$A1$A2$A3.super(a3); this.b3 = b3; }
}}}
```

Note that, since an enclosing instance corresponding to A1.this is included in an enclosing instance corresponding to A2.this, the B3 constructor takes only one extra argument for enclosing instances. Here is C1.C2.C3 class:

```
class C1 extends ... { ...
  class C2 extends ... { ...
    class C3 extends B1.B2.B3 {
      Object c3;
      C3(Object a3, A1.A2 this$A1$A2$A3,
         Object b3, B1.B2 this$B1$B2$B3, Object c3) {
        this$B1$B2$B3.super(a3, this$A1$A2$A3, b3); this.c3 = c3; }
}}}
```

Since the constructor of a superclass `B1.B2.B3` initializes `A2.this`, the constructor `C3` initializes only `B2.this` by qualifying the `super` invocation; the argument `this$A1$A2$A3` is just passed to `super` as an ordinary argument.

In FJI, we restrict the qualification of `super` to be a constructor argument, whereas Java allows any expression for the qualification. This permits the same clean definition of operational semantics we saw in FJ, since all the state information (including fields and enclosing instances) of an object appears in its `new` expression. Moreover, for technical reasons connected with the name mangling involved in the translation semantics, we require that a constructor argument used for qualification of `super` be named $\mathtt{this\$C}_1\mathtt{\$}\cdots\mathtt{\$C}_n$, where $\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n$ is the (direct) superclass, as in the example above.

Lastly, we can now explain why it is not allowed for a class to extend one of its (direct or indirect) inner classes. It is because there is no sensible way to make an instance of such a class. Suppose we could define the class below:

```
class Foo extends Foo.Bar {
  Foo (Foo f) { f.super(); }
  class Bar { ... } }
```

Since `Foo` extends `Foo.Bar`, the constructor `Foo` will need an instance of `Foo` as an argument, making it impossible to make an instance of `Foo`. (Perhaps one could use `null` as the enclosing instance in this case, but this would not be useful, since inner classes are usually supposed to make use of enclosing instances.)

3.3 Syntax

Now, we proceed to the formal definitions of FJI. The abstract syntax of the language is shown at the top left of Figure 2. We use the same notational conventions as in the previous section. The metavariables `S`, `T`, and `U` ranges over types, which are qualified class names (a sequence of simple names $\mathtt{C}_1,\ldots,\mathtt{C}_n$ concatenated by periods). For compactness in the definitions, we introduce the notation $\star$ for a "null qualification" and identify $\star.\mathtt{C}$ with `C`. The metavariable `P` ranges over types (`T`) and $\star$. We write $\mathtt{C} \in \mathtt{P}$ if $\mathtt{P} = \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n$ and $\mathtt{C} = \mathtt{C}_i$ for some i.

A class declaration `L` includes declarations of its simple name `C`, superclass `T`, fields $\overline{\mathtt{T}}$ $\overline{\mathtt{f}}$, constructor `K`, inner classes $\overline{\mathtt{L}}$, and methods $\overline{\mathtt{M}}$. There are two kinds of constructor declaration, depending on whether the superclass is inner or top-level: when the superclass is inner, the subclass constructor must call the `super` constructor with a qualification "`f.`" to provide the enclosing instance visible from the superclass's methods. As we will see in typing rules, constructor arguments should be arranged in the following order: (1) the superclass's fields, initialized by $\mathtt{super(}\overline{\mathtt{f}}\mathtt{)}$ (or $\mathtt{f.super(}\overline{\mathtt{f}}\mathtt{)}$); (2) the enclosing instance of the superclass (if needed); and (3) the fields of the class to be defined, initialized by $\mathtt{this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}$. Like FJ, the body of a method just returns an expression, which is a variable, field access, method invocation, or object instantiation. We assume that the set of variables includes the special variables `this` and `C.this` for every `C`, and that these variables are never used as the names of arguments to methods.

Syntax:

$$\mathtt{T} ::= \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n$$

$$\mathtt{L} ::= \mathtt{class\ C}\triangleleft\mathtt{T\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{L}}\ \overline{\mathtt{M}}\mathtt{\}}$$

$$\begin{array}{rl} \mathtt{K} ::= & \mathtt{C(}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\ \{} \\ & \quad \mathtt{super(}\overline{\mathtt{f}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\}} \\ \mid & \mathtt{C(}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\ \{} \\ & \quad \mathtt{f.super(}\overline{\mathtt{f}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\}} \end{array}$$

$$\mathtt{M} ::= \mathtt{T\ m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}}$$

$$\begin{array}{rl} \mathtt{e} ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m(}\overline{\mathtt{e}}\mathtt{)} \\ \mid & \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \mid \mathtt{e.new<T>\ C(}\overline{\mathtt{e}}\mathtt{)} \end{array}$$

Computation:

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i}$$

$$\frac{\mathit{fields}(\mathtt{T.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{e}_0\mathtt{.new<T>\ C(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i}$$

$$\frac{\begin{array}{c} \mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{d}_0,\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n) \\ \mathtt{c}_n \overset{\text{def}}{=} \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \\ \mathtt{c}_i \overset{\text{def}}{=} \mathit{encl}_{\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_{i+1}}(\mathtt{c}_{i+1})\ ^{i\in 1\ldots n-1} \end{array}}{\begin{array}{c} \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)} \\ \longrightarrow \begin{bmatrix} \overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{c}_n/\mathtt{this}, \\ \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}\ ^{i\in 1\ldots n} \end{bmatrix} \mathtt{d}_0 \end{array}}$$

$$\frac{\begin{array}{c} \mathit{mbody}(\mathtt{m},\mathtt{T.C}) = (\overline{\mathtt{x}},\mathtt{d}_0,\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n) \\ \mathtt{c}_n \overset{\text{def}}{=} \mathtt{e}_0\mathtt{.new<T>\ C(}\overline{\mathtt{e}}\mathtt{)} \\ \mathtt{c}_i \overset{\text{def}}{=} \mathit{encl}_{\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_{i+1}}(\mathtt{c}_{i+1})\ ^{i\in 1\ldots n-1} \end{array}}{\begin{array}{c} \mathtt{e}_0\mathtt{.new<T>\ C(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)} \\ \longrightarrow \begin{bmatrix} \overline{\mathtt{d}}/\overline{\mathtt{x}},\ \mathtt{c}_n/\mathtt{this}, \\ \mathtt{c}_i/\mathtt{C}_i\mathtt{.this}\ ^{i\in 1\ldots n} \end{bmatrix} \mathtt{d}_0 \end{array}}$$

Subtyping:

$$\mathtt{T} <: \mathtt{T} \qquad \frac{\mathtt{S} <: \mathtt{T} \qquad \mathtt{T} <: \mathtt{U}}{\mathtt{S} <: \mathtt{U}}$$

$$\frac{\mathit{CT}(\mathtt{S}) = \mathtt{class\ C}\triangleleft\mathtt{T\ \{...\}}}{\mathtt{S} <: \mathtt{T}}$$

Expression typing:

$$\frac{\Gamma(\mathtt{x}) = \mathtt{T}}{\Gamma \vdash \mathtt{x} \in \mathtt{T}}$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{fields}(\mathtt{T}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0\mathtt{.f}_i \in \mathtt{T}_i}$$

$$\frac{\begin{array}{c} \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{mtype}(\mathtt{m},\mathtt{T}_0) = \overline{\mathtt{U}}{\to}\mathtt{U}_0 \\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} <: \overline{\mathtt{U}} \end{array}}{\Gamma \vdash \mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{U}_0}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \overline{\mathtt{S}} <: \overline{\mathtt{T}}}{\Gamma \vdash \mathtt{new\ C(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{C}}$$

$$\frac{\begin{array}{ccc} \Gamma \vdash \mathtt{e}_0 \in \mathtt{S} & \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} & \\ \mathit{fields}(\mathtt{T.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} & \mathtt{S} <: \mathtt{T} & \overline{\mathtt{S}} <: \overline{\mathtt{T}} \end{array}}{\Gamma \vdash \mathtt{e}_0\mathtt{.new<T>\ C(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{T.C}}$$

Method typing:

$$\frac{\begin{array}{c} \begin{array}{l} \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n, \\ \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_i\ ^{i\in 1\ldots n} \end{array} \vdash \mathtt{e}_0 \in \mathtt{S}_0 \\ \mathit{CT}(\mathtt{C}_1\mathtt{.}\cdots\mathtt{.C}_n) = \mathtt{class\ C}_n\triangleleft\mathtt{S\ \{...\}} \\ \mathtt{S}_0 <: \mathtt{T}_0 \qquad \begin{array}{l} \text{if } \mathit{mtype}(\mathtt{m},\mathtt{S}) = \overline{\mathtt{U}}{\to}\mathtt{U}_0, \\ \text{then } \overline{\mathtt{U}} = \overline{\mathtt{T}} \text{ and } \mathtt{U}_0 = \mathtt{T}_0 \end{array} \end{array}}{\mathtt{T}_0\ \mathtt{m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e}_0\mathtt{;\}\ OK\ IN\ C}_1\mathtt{.}\cdots\mathtt{.C}_n}$$

Class typing:

$$\frac{\begin{array}{c} \mathtt{K} = \begin{array}{l} \mathtt{C(}\overline{\mathtt{S}}\ \overline{\mathtt{g}}\mathtt{,\ }\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{} \\ \quad \mathtt{super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\}} \end{array} \\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{S}}\ \overline{\mathtt{g}} \qquad \mathtt{C} \notin \mathtt{P} \\ \overline{\mathtt{M}}\mathtt{\ OK\ in\ P.C} \qquad \overline{\mathtt{L}}\mathtt{\ OK\ in\ P.C} \end{array}}{\mathtt{class\ C}\triangleleft\mathtt{D\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{L}}\ \overline{\mathtt{M}}\mathtt{\}\ OK\ IN\ P}}$$

$$\frac{\begin{array}{c} \mathtt{K} = \begin{array}{l} \mathtt{C(}\overline{\mathtt{S}}\ \overline{\mathtt{g}}\mathtt{,\ T\ g}_0\mathtt{,\ }\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{} \\ \quad \mathtt{g}_0\mathtt{.super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\}} \end{array} \\ \mathit{fields}(\mathtt{T.D}) = \overline{\mathtt{S}}\ \overline{\mathtt{g}} \qquad \mathtt{C} \notin \mathtt{P} \\ \overline{\mathtt{M}}\mathtt{\ OK\ in\ P.C} \qquad \overline{\mathtt{L}}\mathtt{\ OK\ in\ P.C} \end{array}}{\mathtt{class\ C}\triangleleft\mathtt{T.D\{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{L}}\ \overline{\mathtt{M}}\mathtt{\}\ OK\ IN\ P}}$$

Fig. 2. FJI: Main Definitions

A *program* is a pair of a class table CT (a mapping from types `T` to class declarations `L`) and an expression `e`. `Object` is treated exactly in the same way as in FJ. From the class table, we can read off the subtype relation between classes. We write `S <: T` when `S` is a subtype of `T`—the reflexive and transitive closure of the immediate subclass relation given by the `extends` clauses in CT. This relation is defined formally at the bottom left Figure 2.

We impose the following sanity conditions on the class table: (1) $CT(\texttt{P.C}) = $ `class C ...` for every $\texttt{P.C} \in \mathit{dom}(CT)$. (2) If $CT(\texttt{P.C})$ has an inner class declaration `L` of name `D`, then $CT(\texttt{P.C.D}) = \texttt{L}$. (3) $\texttt{Object} \notin \mathit{dom}(CT)$. (4) For every type `T` (except `Object`) appearing anywhere in CT, we have $\texttt{T} \in \mathit{dom}(CT)$. (5) For every $\texttt{e}_0\texttt{.new<T> C(}\overline{\texttt{e}}\texttt{)}$ (and $\texttt{new C(}\overline{\texttt{e}}\texttt{)}$, resp.) appearing anywhere in CT, we have $\texttt{T.C} \in \mathit{dom}(CT)$ (and $\texttt{C} \in \mathit{dom}(CT)$, resp.). (6) There are no cycles in the subtyping relation. (7) $\texttt{T} \not<: \texttt{T.U}$, for any two types `T` and `T.U`. By conditions (1) and (2), a class table of FJI can be identified with a set of top-level classes. Condition (7) prohibits a class from extending one of its inner classes.

3.4 Auxiliary Functions

For the typing and reduction rules, we need a few auxiliary definitions, given in Figure 3. The fields of a class `T`, written *fields*(`T`), is a sequence $\overline{\texttt{T}}\ \overline{\texttt{f}}$ pairing the class of each field with its name, for all the fields declared in class `T` and all of its superclasses. In addition, *fields*(`T`) collects the types of (direct) enclosing instances of all the superclasses of `T`. For example, *fields*(`C1.C2.C3`) returns the following sequence:

$$\begin{array}{lll}
\mathit{fields}(\texttt{C1.C2.C3}) = & \texttt{Object a3,} & \text{(field from } \texttt{A1.A2.A3}\text{)} \\
& \texttt{A1.A2 this\$A1\$A2\$A3,} & \text{(enclosing instance } \texttt{A2.this}\text{)} \\
& \texttt{Object b3,} & \text{(field from } \texttt{B1.B2.B3}\text{)} \\
& \texttt{B1.B2 this\$B1\$B2\$B2,} & \text{(enclosing instance } \texttt{B2.this}\text{)} \\
& \texttt{Object c3} & \text{(field from } \texttt{C1.C2.C3}\text{)}
\end{array}$$

The third rule in the definition inserts enclosing instance information between the fields $\overline{\texttt{S}}\ \overline{\texttt{g}}$ of the superclass `U.D` and the fields $\overline{\texttt{T}}\ \overline{\texttt{f}}$ of the current class. In a well-typed program, *fields*(`T`) will always agree with the constructor argument list of `T`.

The type of the method `m` in class `T`, written *mtype*(`m`, `T`), is a pair, written $\overline{\texttt{S}}\rightarrow\texttt{S}$, of a sequence of argument types $\overline{\texttt{S}}$ and a result type `S`. Similarly, the body of the method `m` in class `T`, written *mbody*(`m`, `T`), is a triple, written ($\overline{\texttt{x}}$, `e`, `T`), of a sequence of parameters $\overline{\texttt{x}}$, an expression `e`, and a class `T` where the method is defined.

The function $\mathit{encl}_{\texttt{T}}(\texttt{e})$ plays a crucial role in the semantics of FJI. Intuitively, when `e` is a top-level or inner class instantiation, $\mathit{encl}_{\texttt{T}}(\texttt{e})$ returns the direct enclosing instance of `e` that is visible from class `T` (i.e., the enclosing instance that provides the correct lexical environment for methods inherited from `T`). The first rule is the simplest case: since the type of an expression $\texttt{e}_0\texttt{.new<T> C(}\overline{\texttt{e}}\texttt{)}$ agrees with the subscript `T.C`, it just returns the (direct) enclosing instance

Field lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\begin{array}{c} CT(\texttt{T}) = \texttt{class C}\triangleleft\texttt{D \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\ \mathit{fields}(\texttt{D}) = \overline{\texttt{S}}\ \overline{\texttt{g}} \end{array}}{\mathit{fields}(\texttt{T}) = \overline{\texttt{S}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}}}$$

$$\frac{\begin{array}{c} CT(\texttt{T}) = \texttt{class C}\triangleleft\texttt{U.D \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\ \mathit{fields}(\texttt{U.D}) = \overline{\texttt{S}}\ \overline{\texttt{g}} \\ \texttt{U} = \texttt{C}_1\texttt{.}\cdots\texttt{.C}_n \\ \texttt{f}_0 = \texttt{this\$C}_1\texttt{\$}\cdots\texttt{\$C}_n\texttt{\$D} \end{array}}{\mathit{fields}(\texttt{T}) = \overline{\texttt{S}}\ \overline{\texttt{g}}, \texttt{U f}_0, \overline{\texttt{T}}\ \overline{\texttt{f}}}$$

Method type lookup:

$$\frac{\begin{array}{c} CT(\texttt{T}) = \texttt{class C}\triangleleft\texttt{S \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{U}_0\texttt{ m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}} \in \overline{\texttt{M}} \end{array}}{\mathit{mtype}(\texttt{m}, \texttt{T}) = \overline{\texttt{U}}\to\texttt{U}_0}$$

$$\frac{\begin{array}{c} CT(\texttt{T}) = \texttt{class C}\triangleleft\texttt{S \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{m} \text{ is not defined in } \overline{\texttt{M}} \end{array}}{\mathit{mtype}(\texttt{m}, \texttt{T}) = \mathit{mtype}(\texttt{m}, \texttt{S})}$$

Method body lookup:

$$\frac{\begin{array}{c} CT(\texttt{T}) = \texttt{class C}\triangleleft\texttt{S \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{U}_0\texttt{ m (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}} \in \overline{\texttt{M}} \end{array}}{\mathit{mbody}(\texttt{m}, \texttt{T}) = (\overline{\texttt{x}}, \texttt{e}, \texttt{T})}$$

$$\frac{\begin{array}{c} CT(\texttt{T}) = \texttt{class C}\triangleleft\texttt{S \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}} \\ \texttt{m} \text{ is not defined in } \overline{\texttt{M}} \end{array}}{\mathit{mbody}(\texttt{m}, \texttt{T}) = \mathit{mbody}(\texttt{m}, \texttt{S})}$$

Enclosing instance lookup:

$$\mathit{encl}_{\texttt{T.C}}(\texttt{e}_0\texttt{.new<T> C(}\overline{\texttt{e}}\texttt{)}) = \texttt{e}_0$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C}\triangleleft\texttt{D \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;...\}} \\ \#(\overline{\texttt{f}}) = \#(\overline{\texttt{e}}) \end{array}}{\begin{array}{c} \mathit{encl}_{\texttt{T}}(\texttt{new C(}\overline{\texttt{d}}\texttt{, }\overline{\texttt{e}}\texttt{)}) \\ = \mathit{encl}_{\texttt{T}}(\texttt{new D(}\overline{\texttt{d}}\texttt{)}) \end{array}}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C}\triangleleft\texttt{U.D \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;...\}} \\ \#(\overline{\texttt{f}}) = \#(\overline{\texttt{e}}) \end{array}}{\begin{array}{c} \mathit{encl}_{\texttt{T}}(\texttt{new C(}\overline{\texttt{d}}\texttt{, d}_0\texttt{, }\overline{\texttt{e}}\texttt{)}) \\ = \mathit{encl}_{\texttt{T}}(\texttt{d}_0\texttt{.new<U> D(}\overline{\texttt{d}}\texttt{)}) \end{array}}$$

$$\frac{\begin{array}{c} CT(\texttt{S.C}) = \texttt{class C}\triangleleft\texttt{D \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;...\}} \\ \#(\overline{\texttt{f}}) = \#(\overline{\texttt{e}}) \qquad \texttt{T} \neq \texttt{S.C} \end{array}}{\begin{array}{c} \mathit{encl}_{\texttt{T}}(\texttt{e}_0\texttt{.new<S> C(}\overline{\texttt{d}}\texttt{, }\overline{\texttt{e}}\texttt{)}) \\ = \mathit{encl}_{\texttt{T}}(\texttt{new D(}\overline{\texttt{d}}\texttt{)}) \end{array}}$$

$$\frac{\begin{array}{c} CT(\texttt{S.C}) = \texttt{class C}\triangleleft\texttt{U.D \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;...\}} \\ \#(\overline{\texttt{f}}) = \#(\overline{\texttt{e}}) \qquad \texttt{T} \neq \texttt{S.C} \end{array}}{\begin{array}{c} \mathit{encl}_{\texttt{T}}(\texttt{e}_0\texttt{.new<S> C(}\overline{\texttt{d}}\texttt{, d}_0\texttt{, }\overline{\texttt{e}}\texttt{)}) \\ = \mathit{encl}_{\texttt{T}}(\texttt{d}_0\texttt{.new<U> D(}\overline{\texttt{d}}\texttt{)}) \end{array}}$$

Fig. 3. FJI: Auxiliary definitions

$\texttt{e}_0$. The other rules follow a common pattern; we explain the fifth rule as a representative. Since the subscripted type `T` is different from the type of the argument $\texttt{e}_0\texttt{.new<S> C(}\overline{\texttt{d}}\texttt{, d}_0\texttt{, }\overline{\texttt{e}}\texttt{)}$, the enclosing instance $\texttt{e}_0$ is not the correct answer. We therefore make a recursive call with an object $\texttt{d}_0\texttt{.new<U> D(}\overline{\texttt{d}}\texttt{)}$ of the superclass obtained by dropping $\texttt{e}_0$ and as many arguments $\overline{\texttt{e}}$ as the fields $\overline{\texttt{f}}$ of the class `S.C`. We keep going like this until, finally, the argument becomes an instance of `T` and we match the first rule. For example:

$$\begin{aligned} & \mathit{encl}_{\texttt{A1.A2.A3}}(\texttt{e}_0\texttt{.new<C1.C2> C3(a, e}_1\texttt{, b, e}_2\texttt{, c)}) \\ = {} & \mathit{encl}_{\texttt{A1.A2.A3}}(\texttt{e}_2\texttt{.new<B1.B2> B3(a, e}_1\texttt{ ,b)}) \\ = {} & \mathit{encl}_{\texttt{A1.A2.A3}}(\texttt{e}_1\texttt{.new<A1.A2> A3(a)}) \\ = {} & \texttt{new A1().new<A1> A2()} \end{aligned}$$

where $\texttt{e}_1 = \texttt{new A1().new<A1> A2()}$ and $\texttt{e}_2 = \texttt{new B1().new<B1> B2()}$.

Note that the *encl* function outputs only the direct enclosing instance. To obtain outer enclosing instances, such as `A1.this`, *encl* can be used repeatedly: $\mathit{encl}_{\mathtt{A1.A2}}(\mathit{encl}_{\mathtt{A1.A2.A3}}(\mathtt{e}))$.

3.5 Computation

As in FJ, the reduction relation of FJI has the form $\mathtt{e} \longrightarrow \mathtt{e}'$. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. The reduction rules are given in the middle of the left column of Figure 2. There are four reduction rules, two for field access and two for method invocation. The field access expression $\mathtt{new\ C(\overline{e}).f}_i$ looks up the field names $\overline{\mathtt{f}}$ of `C` using *fields*(`C`) and yields the constructor argument $\mathtt{e}_i$ in the position corresponding to $\mathtt{f}_i$ in the field list; $\mathtt{e}_0\mathtt{.new\text{<}T\text{>}\ C(\overline{e}).f}_i$ behaves similarly. The method invocation expression $\mathtt{new\ C(\overline{e}).m(\overline{d})}$ first calls *mbody*(`m`, `C`) to obtain a triple of the sequence of formal arguments $\overline{\mathtt{x}}$, the method body `e`, and the class $\mathtt{C}_1.\cdots.\mathtt{C}_n$ where `m` is defined; it yields a substitution instance of the method body in which the $\overline{\mathtt{x}}$ are replaced with the actual arguments $\overline{\mathtt{d}}$, the special variables `this` and $\mathtt{C}_n\mathtt{.this}$ with the receiver object $\mathtt{new\ C(\overline{e})}$, and each $\mathtt{C}_i\mathtt{.this}$ (for $i < n$) with the corresponding enclosing instance $\mathtt{c}_i$, obtained from *encl*. Since the method to be invoked is defined in $\mathtt{C}_1.\cdots.\mathtt{C}_n$, the direct enclosing instance $\mathtt{C}_{n-1}\mathtt{.this}$ is obtained by $\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e})$, where `e` is the receiver object; similarly, $\mathtt{C}_{n-2}\mathtt{.this}$ is obtained by $\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_{n-1}}(\mathit{encl}_{\mathtt{C}_1.\cdots.\mathtt{C}_n}(\mathtt{e}))$, and so on. The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules (if $\mathtt{e} \longrightarrow \mathtt{e}'$ then $\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}$, and the like), which we omit here.

For example, if the class table includes `Outer`, `RefinedInner`, `Pair`, `A`, and `B`, then

```
new RefinedInner(
 new Outer(new Pair(new A(), new B())), new Object()).snd_p()
```

reduces to `new B()` as follows:

```
   new RefinedInner(
    new Outer(new Pair(new A(), new B())), new Object()).snd_p()
⟶ new Outer(new Pair(new A(), new B())).p.snd
⟶ new Pair(new A(), new B()).snd
⟶ new B()
```

3.6 Typing Rules

The typing rules for expressions, method declarations, and class declarations are given in the right column of Figure 2. An environment Γ is a finite mapping from variables to types, written $\overline{\mathtt{x}}\mathtt{:}\overline{\mathtt{T}}$. The typing judgment for expressions has the form $\Gamma \vdash \mathtt{e} \in \mathtt{T}$, read "in the environment Γ, expression `e` has type `T`." The typing rules are syntax directed, with one rule for each form of expression. The typing rules for object instantiations and method invocations check that each actual parameter has a type which is a subtype of the corresponding formal parameter type obtained by *fields* or *mtype*; the enclosing object must have a type which is a subtype of the annotated type $\mathtt{T}$ in $\mathtt{new<T>}$.

The typing judgment for method declarations has the form $\mathtt{M\ OK\ IN\ C_1.\cdots.C}_n$, read "method declaration $\mathtt{M}$ is ok if it is declared in class $\mathtt{C_1.\cdots.C}_n$." The body of the method is typed under the context in which the formal parameters of the method have their declared types and each $\mathtt{C}_i\mathtt{.this}$ has the type $\mathtt{C_1.\cdots.C}_i$. If a method with the same name is declared in the superclass then it must have the same type in the subclass.

The typing judgment for class declarations has the form $\mathtt{L\ OK\ IN\ P}$, read "class declaration $\mathtt{L}$ is ok if it is declared in $\mathtt{P}$." If $\mathtt{P}$ is a type $\mathtt{T}$, the class declaration $\mathtt{L}$ is an inner class; otherwise, $\mathtt{L}$ is a top-level class. The typing rules check that the constructor applies `super` to the fields of the superclass and initializes the fields declared in this class, and that each method declaration and inner class declaration in the class is ok. The condition $\mathtt{C} \notin \mathtt{P}$ ensures that the (simple) class name to be defined is not also a simple name of one of the enclosing classes, so as to avoid ambiguity of the meaning of `C.this`.

3.7 Properties

As well as FJ programs, FJI programs also enjoy standard subject reduction and progress properties, which together guarantee that well-typed programs never get stuck on field accesses or method invocations.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash \mathtt{e} \in \mathtt{T}$ *and* $\mathtt{e} \longrightarrow \mathtt{e}'$, *then* $\Gamma \vdash \mathtt{e}' \in \mathtt{T}'$ *for some* $\mathtt{T}'$ *such that* $\mathtt{T}' <: \mathtt{T}$.

Theorem 2 (Progress). *Suppose* $\mathtt{e}$ *is a well-typed expression.*

(1) If $\mathtt{e}$ *includes* $\mathtt{new\ C_0(\overline{e}).f}$ *as a subexpression, then* $\mathit{fields}(\mathtt{C_0}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *and* $\mathtt{f} \in \overline{\mathtt{f}}$. *Similarly, if* $\mathtt{e}$ *includes* $\mathtt{e_0.new<T_0>\ C(\overline{e}).f}$ *as a subexpression, then* $\mathit{fields}(\mathtt{T_0.C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}$ *and* $\mathtt{f} \in \overline{\mathtt{f}}$.

(2) If $\mathtt{e}$ *includes* $\mathtt{new\ C_0(\overline{e}).m(\overline{d})}$ *as a subexpression, then* $\mathit{mbody}(\mathtt{m}, \mathtt{C_0}) = (\overline{\mathtt{x}}, \mathtt{e_0}, \mathtt{C_1.\cdots.C}_n)$ *and* $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$, *and* $\mathtt{c}_1, \ldots, \mathtt{c}_n$ *appearing in the third computation rule are well defined.*
Similarly, if $\mathtt{e}$ *includes* $\mathtt{e_0.new<T_0>\ C(\overline{e}).m(\overline{d})}$ *as a subexpression, then* $\mathit{mbody}(\mathtt{m}, \mathtt{T_0.C}) = (\overline{\mathtt{x}}, \mathtt{d_0}, \mathtt{C_1.\cdots.C}_n)$ *and* $\#(\overline{\mathtt{x}}) = \#(\overline{\mathtt{d}})$ *and* $\mathtt{c}_1, \ldots, \mathtt{c}_n$ *appearing in the fourth computation rule are well defined.*

4 Translation Semantics

In this section we consider the other style of semantics: translation from FJI to FJ. Every inner class is compiled to a top-level class with one additional field holding a reference to the direct enclosing instance; occurrences of qualified `this` are translated into accesses to this field. For example, the `Outer` and `RefinedInner` classes in the previous section are compiled to the following three FJ classes.

```
class Outer extends Object {
  Pair p;
  Outer(Pair p) { super(); this.p = p; }
  Outer$Inner make_inner () { return new Outer$Inner(this); }
}

class Outer$Inner extends Object {
  Outer this$Outer$Inner;
  Outer$Inner(Outer this$Outer$Inner) {
    super(); this.this$Outer$Inner = this$Outer$Inner; }
  Object snd_p { return this.this$Outer$Inner.p.snd; }
}

class RefinedInner extends Outer$Inner {
  Object c;
  RefinedInner(Outer this$Outer$Inner, Object c) {
    super(this$Outer$Inner); this.c = c;
  }
}
```

The inner class `Outer.Inner` is compiled to the top-level class `Outer$Inner`; the field `this$Outer$Inner` holds an `Outer` object, which corresponds to the direct enclosing instance `Outer.this` in the original FJI program; thus, `Outer.this` is compiled to the field access expression `this.this$Outer$Inner`.

We give a compilation function $|\cdot|$ for each syntactic category. Except for types, the compilation functions take as their second argument the FJI class name (or, $\star$) where the entity being translated is defined, written $|\cdot|_{\mathtt{T}}$ (or $|\cdot|_{\star}$).

4.1 Types, Expressions and Methods

Every qualified class name is translated to a simple name obtained by syntactic replacement of `.` with `$`.

$$|\mathtt{C}_1.\cdots.\mathtt{C}_n| \;=\; \mathtt{C}_1\mathtt{\$}\cdots\mathtt{\$C}_n$$

The compilation of expressions, written $|\mathtt{e}|_{\mathtt{T}}$, is given below. We write $|\overline{\mathtt{e}}|_{\mathtt{T}}$ as shorthand for $|\mathtt{e}_1|_{\mathtt{T}},\ldots,|\mathtt{e}_n|_{\mathtt{T}}$ (and similarly for $|\overline{\mathtt{T}}|$, $|\overline{\mathtt{M}}|_{\mathtt{T}}$ and $|\overline{\mathtt{L}}|_{\mathtt{P}}$).

$$\begin{array}{ll}
|\mathtt{x}|_{\mathtt{T}} & = \mathtt{x} \\
|\mathtt{e}_0\mathtt{.f}|_{\mathtt{T}} & = |\mathtt{e}_0|_{\mathtt{T}}\mathtt{.f} \\
|\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)}|_{\mathtt{T}} & = |\mathtt{e}_0|_{\mathtt{T}}\mathtt{.m(}\,|\overline{\mathtt{e}}|_{\mathtt{T}}\,\mathtt{)} \\
|\mathtt{new\ D(}\overline{\mathtt{e}}\mathtt{)}|_{\mathtt{T}} & = \mathtt{new\ D(}\,|\overline{\mathtt{e}}|_{\mathtt{T}}\,\mathtt{)} \\
|\mathtt{e}_0\mathtt{.new<T>\ D(}\overline{\mathtt{e}}\mathtt{)}|_{\mathtt{T}} & = \mathtt{new}\ \ |\mathtt{T.D}|\ \mathtt{(}\,|\overline{\mathtt{e}}|_{\mathtt{T}}\,\mathtt{,}\ |\mathtt{e}_0|_{\mathtt{T}}\,\mathtt{)} \\
|\mathtt{this}|_{\mathtt{T}} & = \mathtt{this} \\
|\mathtt{C}_n\mathtt{.this}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} & = \mathtt{this} \\
|\mathtt{C}_i\mathtt{.this}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} & = |\mathtt{C}_{i+1}\mathtt{.this}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n}\mathtt{.this\$C}_1\mathtt{\$}\cdots\mathtt{\$C}_{i+1} \quad (1 \le i \le n-1)
\end{array}$$

As we saw above, a compiled inner class has one additional field, called `this$` $|\mathtt{T}|$, where `T` is the original class name. $\mathtt{C}_i\mathtt{.this}$ in the class $\mathtt{C}_1.\cdots.\mathtt{C}_n$ becomes an expression that follows references to the direct enclosing instance in sequence until it reaches the desired one. An enclosing instance $\texttt{e}_0$ of $\texttt{e}_0\texttt{.new<T> C(}\overline{\texttt{e}}\texttt{)}$ will become the last argument of the compiled constructor invocation.

Compilation of methods, written $|\texttt{M}|_{\texttt{T}}$, is straightforward. We use the notation $\left|\overline{\texttt{T}}\right|\ \overline{\texttt{x}}$ for $|\texttt{T}_1|\ \texttt{x}_1, \dots, |\texttt{T}_n|\ \texttt{x}_n$.

$$\left|\texttt{T}_0\ \texttt{m (}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{) \{ return e; \}}\right|_{\texttt{T}} = |\texttt{T}_0|\ \texttt{m(}\left|\overline{\texttt{T}}\right|\ \overline{\texttt{x}}\texttt{) \{ return }|\texttt{e}|_{\texttt{T}}\texttt{; \}}$$

4.2 Constructors and Classes

Compilation of constructors, written $|\texttt{K}|_{\texttt{T}}$, is given below.

$$\left|\begin{array}{l}\texttt{C(}\overline{\texttt{S}}\ \overline{\texttt{g}}\texttt{, }\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)}\\ \quad\texttt{\{ super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}}\end{array}\right|_{\texttt{C}} = \begin{array}{l}\texttt{C(}\left|\overline{\texttt{S}}\right|\ \overline{\texttt{g}}\texttt{, }\left|\overline{\texttt{T}}\right|\ \overline{\texttt{f}}\texttt{)}\\ \quad\texttt{\{super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;\}}\end{array}$$

$$\left|\begin{array}{l}\texttt{C(}\overline{\texttt{S}}\ \overline{\texttt{g}}\texttt{, S}_0\ \texttt{g}_0\texttt{, }\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)}\\ \quad\texttt{\{ g}_0\texttt{.super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}}\end{array}\right|_{\texttt{C}} = \begin{array}{l}\texttt{C(}\left|\overline{\texttt{S}}\right|\ \overline{\texttt{g}}\texttt{, }|\texttt{S}_0|\ \texttt{g}_0\texttt{, }\left|\overline{\texttt{T}}\right|\ \overline{\texttt{f}}\texttt{)}\\ \quad\texttt{\{super(}\overline{\texttt{g}}\texttt{, g}_0\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;\}}\end{array}$$

$$\left|\begin{array}{l}\texttt{C(}\overline{\texttt{S}}\ \overline{\texttt{g}}\texttt{, }\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)}\\ \quad\texttt{\{ super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}}\end{array}\right|_{\texttt{T.C}} = \begin{array}{l}|\texttt{T.C}|\texttt{(}\left|\overline{\texttt{S}}\right|\ \overline{\texttt{g}}\texttt{, }\left|\overline{\texttt{T}}\right|\ \overline{\texttt{f}}\texttt{,}\\ \qquad\qquad |\texttt{T}|\ \texttt{this\$}|\texttt{T.C}|\texttt{)}\\ \texttt{\{super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{;}\\ \quad\texttt{this.this\$}|\texttt{T.C}|\texttt{=this\$}|\texttt{T.C}|\texttt{;\}}\end{array}$$

$$\left|\begin{array}{l}\texttt{C(}\overline{\texttt{S}}\ \overline{\texttt{g}}\texttt{, S}_0\ \texttt{g}_0\texttt{, }\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{)}\\ \quad\texttt{\{ g}_0\texttt{.super(}\overline{\texttt{g}}\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{; \}}\end{array}\right|_{\texttt{T.C}} = \begin{array}{l}|\texttt{T.C}|\texttt{(}\left|\overline{\texttt{S}}\right|\ \overline{\texttt{g}}\texttt{, }|\texttt{S}_0|\ \texttt{g}_0\texttt{, }\left|\overline{\texttt{T}}\right|\ \overline{\texttt{f}}\texttt{,}\\ \qquad\qquad |\texttt{T}|\ \texttt{this\$}|\texttt{T.C}|\texttt{)}\\ \texttt{\{super(}\overline{\texttt{g}}\texttt{, g}_0\texttt{); this.}\overline{\texttt{f}}\texttt{ = }\overline{\texttt{f}}\texttt{;}\\ \quad\texttt{this.this\$}|\texttt{T.C}|\texttt{=this\$}|\texttt{T.C}|\texttt{;\}}\end{array}$$

It has four cases, depending on whether the current class is a top-level class or an inner class and whether its superclass is a top-level class or an inner class. When the current class is an inner class, one more argument corresponding to the enclosing instance is added to the argument list; the name of the constructor becomes $|\texttt{T.C}|$, the translation of the qualified name of the class. When the superclass is inner (the third and fourth cases), the argument used for the qualification of $\texttt{f.super(}\overline{\texttt{f}}\texttt{)}$ becomes the last argument of the $\texttt{super()}$ invocation.

Finally, the compilation of classes, written $|\texttt{L}|_{\texttt{P}}$, is as follows:

$$\left|\texttt{class C}\lhd\texttt{S \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}}\right|_{\star} = \texttt{class C}\lhd|\texttt{S}|\ \texttt{\{}\left|\overline{\texttt{T}}\right|\ \overline{\texttt{f}}\texttt{; }|\texttt{K}|_{\texttt{C}}\ \left|\overline{\texttt{M}}\right|_{\texttt{C}}\texttt{\}}\ \left|\overline{\texttt{L}}\right|_{\texttt{C}}$$

$$\left|\texttt{class C}\lhd\texttt{S \{}\overline{\texttt{T}}\ \overline{\texttt{f}}\texttt{; K }\overline{\texttt{L}}\ \overline{\texttt{M}}\texttt{\}}\right|_{\texttt{T}} = \begin{array}{l}\texttt{class }|\texttt{T.C}|\lhd|\texttt{S}|\ \texttt{\{}\\ \quad\left|\overline{\texttt{T}}\right|\ \overline{\texttt{f}}\texttt{; }|\texttt{T}|\ \texttt{this\$}|\texttt{T.C}|\texttt{; }|\texttt{K}|_{\texttt{T.C}}\ \left|\overline{\texttt{M}}\right|_{\texttt{T.C}}\texttt{\}}\\ \left|\overline{\texttt{L}}\right|_{\texttt{T.C}}\end{array}$$

The constructor, inner classes, and methods of class $\texttt{C}$ defined in $\texttt{P}$ are compiled with auxiliary argument $\texttt{P.C}$. Inner classes $\overline{\texttt{L}}$ become top-level classes. As in constructor compilation, when the compiled class is inner, its name changes to $|\texttt{T.C}|$ and the field $\texttt{this\$}|\texttt{T.C}|$, holding an enclosing instance, is added. The compilation of the class table, written $|CT|$, is achieved by compiling all top-level classes $\overline{\texttt{L}}$ in CT (i.e., $\left|\overline{\texttt{L}}\right|_{\star}$).

4.3 Properties of Translation Semantics

We develop three theorems here. First, the translation semantics preserves typing, in the sense that a well-typed FJI program is compiled to a well-typed FJ program (Theorem 3). Second, we show that the behavior of a compiled program exactly reflects the behavior of the original program in FJI: for every step of reduction of a well-typed FJI program, the compiled program takes one or more steps and reaches a corresponding state (Theorem 4) and vice versa (Theorem 5).

Theorem 3 (Compilation preserves typing). *When* $\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{T}}$*, we write* $|\Gamma|$ *for* $\overline{\mathtt{x}} : \left|\overline{\mathtt{T}}\right|$*. If an FJI class table CT is ok and* $\overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C}_1.\cdots.\mathtt{C}_n, \mathtt{C}_i\mathtt{.this} : \mathtt{C}_1.\cdots.\mathtt{C}_i\ ^{i\in 1\ldots n} \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ *with respect to CT, then* $|CT|$ *is ok and* $\overline{\mathtt{x}} : \left|\overline{\mathtt{T}}\right|, \mathtt{this} : |\mathtt{C}_1.\cdots.\mathtt{C}_n| \vdash_{\mathrm{FJ}} |\mathtt{e}|_{\mathtt{C}_1.\cdots.\mathtt{C}_n} \in |\mathtt{T}|$ *with respect to* $|CT|$*.*

Theorem 4 (Compilation commutes with reduction). *If* $\Gamma \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ *where* $dom(\Gamma)$ *does not include* `this` *or* `C.this` *for any* `C`*, and* $\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{e}'$*, then* $|\mathtt{e}|_\star \longrightarrow_{\mathrm{FJ}}{}^{+} |\mathtt{e}'|_\star$*.*

Theorem 5 (Compilation preserves termination). *If* $\Gamma \vdash_{\mathrm{FJI}} \mathtt{e} \in \mathtt{T}$ *where* $dom(\Gamma)$ *does not include* `this` *or* `C.this`*, and* $|\mathtt{e}|_\star \longrightarrow_{\mathrm{FJ}} \mathtt{e}'$*, then* $\mathtt{e} \longrightarrow_{\mathrm{FJI}} \mathtt{e}''$ *and* $\mathtt{e}' \longrightarrow_{\mathrm{FJ}}{}^{*} |\mathtt{e}''|_\star$ *for some* $\mathtt{e}''$*.*

Unfortunately, Theorems 4 and 5 would not hold for a call-by-value version of FJI, since their properties depend on our non-deterministic reduction strategy. An intuitive reason is as follows. In FJI, after method invocation, `C.this` is directly replaced with the corresponding enclosing instance. On the other hand, in the compiled FJ program, `C.this` is translated to an expression $\mathtt{this.f}_1\mathtt{.f}_2.\cdots.\mathtt{f}_n$, where each $\mathtt{f}_i$ is a mangled field name, and its evaluation may be guarded by its context. Therefore, reduction steps do not commute with compilation straightforwardly. Nevertheless, it should be possible to show correctness pby using another technique, such as contextual equivalence [18], as Glew proved a similar result in the context of object closure conversion for a call-by-value object calculus [7].

5 Elaboration

In this section we formalize the elaboration of user programs. In user programs, the receivers of field access or method invocation, the enclosing instances of inner class instantiation, and the qualifications of type names may be omitted. For example, a simple name `C` means an inner class `T.C` when it is used in the direct enclosing class `T`. A basic job of elaboration is to find where a name `f`, `m`, or `C` is bound and to recover its receiver information or absolute path.

In the conventional scoping rules of simple block structured languages, simple names are bound to their syntactically nearest declaration. In Java, however, they can be bound to declarations in superclasses, or even in superclasses of enclosing classes. For example, in the class below, `f` in the method `m` is bound to the field `f` of the enclosing class `C` *unless* `D` has a field `f`.

```
class C extends Object {
  Object f; ...
  class D extends Object { ...
    Object m () { return f; }
  }
}
```

Similarly, `f` in the method `m` is bound to the field `f` of its superclass `B` (when neither `C` nor `D` has field `f`) in the following classes.

```
class B extends Object { Object f; ... }
class C extends Object { ...
  class D extends B { ...
    Object m () { return f; }
  }
}
```

In general, beginning with the current class where the field/method name is used, the search algorithm looks for the definition in superclasses; if there is no definition in any superclass, it looks in the direct enclosing class and its superclasses, and then in the second direct enclosing class and its super classes, and so on. Once the declaration where a name is bound is known, it is easy to recover the appropriate qualification. In the examples above, `f` becomes `C.this.f` and `D.this.f`, respectively.

Suppose the algorithm above finds the definition of the field/method in one of the superclasses of the current class. Then, a field/method of the same name must not be defined in any of the enclosing classes. Similarly, if the field/method definition is found in a superclass of an enclosing class `C`, a field/method of the same name must not be defined in any of `C`'s enclosing classes. In the example above, if both `B` and `C` declared a field `f` (and `D` did not), then elaboration would fail as `f` in `m` is *ambiguous*; the user must write `C.this.f` or `D.this.f`, specifying the enclosing instance explicitly. This rule also has one significant exception: it is not considered ambiguous if the definition found in a superclass is *also* the syntactically nearest definition in enclosing classes. This situation occurs when an inner class extends one of its enclosing classes. For example, suppose `E` does not declare the field `f` in the class definition below.

```
class C extends Object {
  Object f; ...
  class D extends Object { ...
    class E extends C { ...
      Object m () { return f; }
    }
  }
}
```

The reference to `f` in `m` is not ambiguous unless `D` declares the field `f`. (The algorithm finds the definition `f` in a *superclass* of `E`.)

Simple type names obey similar elaboration rules. For example, D occurring in C is elaborated to C.D. However, unlike field names and method names, pre-elaborated type names themselves can be qualified. In such a case the head simple name is elaborated first, then it looks up the definitions of the following names in a manner similar to field lookup. For example, consider the following class declarations:

```
class A extends Object { ...
  class B extends Object { ... }
}
class C extends Object { ...
  class D extends A { ... }
}
class E extends C { D.B f; ... }
```

The type D.B of f is elaborated to A.B as follows:

1. The first name D is elaborated to C.D.
2. It is checked whether C.D.B makes sense; in this case, it does not, since the inner class D does not have the declaration of B. The elaborator replaces C.D with its superclass A and elaborates A.B in the context of C.
3. Since A is not declared in C, it denotes the top-level class A.
4. Finally, since B is declared in the top-level class A, A.B is the elaborated type for D.B in the context of E.

Last, we describe how a constructor invocation new T($\overline{e}$) is elaborated. Actually, it is slightly more involved than others since it requires both elaboration of the type and recovering of an enclosing instance (when it turns out to be instantiation of an inner class). First of all, the pre-elaborated type name T is elaborated to T′. If T′ is a simple name C, then the constructor invocation does not need an enclosing instance. On the other hand, if T′ is U.C, then we have to make up an enclosing instance D.this, whose type is subtype of U, by checking which enclosing class is a subclass of U. Finally, among such enclosing classes, the innermost one is chosen and new T($\overline{e}$) is elaborated to D.this.new<U> C(...). The annotation <U> is important to specify which inner class is instantiated, since there might be more than one inner class C defined in classes between D and U. Consider the following classes and the expression new A.B() inside the class D.E:

```
class A extends Object { ...
  class B extends Object { ... }
}
class C extends A { ...
  class B extends Object { ... }
}
class D extends C { ...
  class E extends C { ...
    Object m () { ... new A.B() ...}
  }
}
```

First, `A.B` is elaborated to itself. Now, we need to find out which enclosing class (including the current class) is a subclass of `A`. In this case, both `D` and `D.E` are; then, the innermost one, `D.E`, is chosen, and `new A.B()` is elaborated to `E.this.new<A> B()`. The annotation `<A>` is important since we have to remember that the class `A.B` is to be instantiated (not `C.B`).

For brevity, we omit the formal rules of elaboration, which closely follow the algorithm described above; interested readers are referred to a companion technical report [11].

6 Interpretations of the Inner Class Specification

Through this work, we have experimented a few Java compilers, including Sun's JDK (for Solaris), JDK for linux, and `guavac`. Besides finding a few bugs related to inner classes (mostly already known to the developers), we observed some interesting variations in behavior corresponding to an underspecification in the currently available Inner Classes Specification [12], concerning the meaning of the `C.this` expression. Consider the following Java program:

```
class C {
  void who () {
    System.out.println("I'm a C object");
  }

  class D extends C {
    void m () { C.this.who(); }
    void who () {
      System.out.println("I'm a C.D object");
  }}

  public static void main (String[] args) {
    new C().new D().m();
  }
}
```

Surprisingly, this program prints out `I'm a C.D object` when compiled with JDK 1.1.7a, but `I'm a C object` under JDK 1.2. In the old JDK, the meaning of `C.this` is exactly the same as `D.this` or `this` when `C` is a superclass of the inner class `C.D`; thus, `C.this` is bound to the receiver `new C().new D ()`. In JDK 1.2, on the other hand, `C.this` is always bound to the enclosing object of the receiver regardless of superclass.

7 Related Work

Nested classes in Beta. Beta [15] also allows nested class definitions (as an instance of nested *patterns*, the only abstraction mechanism in Beta, which unifies classes and procedures). There are two significant differences from inner classes. First, inner classes are covariantly specialized in a subclass: for example, if

C <: D and both C and D have the declaration of an inner class of name E, then C.E must extend D.E. Second, nested classes are *virtual* [14], in the sense that it depends on *run-time type* of the enclosing instance which constructor is invoked. A constructor invocation e.new E($\bar{\texttt{e}}$) instantiates an object of class C.E when the run-time type of e is C while it instantiates an object of class D.E when that of e is D.

Madsen has recently described the algorithm of elaboration (they call semantic analysis) used in the Mjølner Beta compiler [13]. The algorithm is very close to the rules presented in Section 5, in a sense that the search order is the same as ours, although the presence of virtual classes complicates the algorithm.

Specification of inner classes. In the currently available Inner Classes Specification [12], semantics of inner classes is given as a translation from inner classes to top-level classes. It also explains how inner classes affect other language aspects, such as synchronization, access restriction and binary compatibility. However, description is rather informal and sometimes vague, resulting in different implementations with different semantics, as explained in the previous section.

Object closure conversion. Recently, Glew [7] has studied closure conversion in the context of a call-by-value object calculus (without classes) and shown correctness of conversion based on contextual equivalence. Our translation semantics can also be viewed as closure conversion of class definitions. Since his calculus does not have classes, semantic account of interaction between inheritance and nested classes is not given.

Microsoft's delegates. Microsoft has proposed *delegates* [16] as an alternative to inner classes. The basic idea of delegates resembles the function pointers found in C and C++. Programmers can create a delegate with an expression of the form e.m (without parameters) and pass it elsewhere; later, the method m can be invoked through the delegate. We believe it would be possible to model delegates in an extension of FJ, as we have done here for inner classes. On the one hand, the formalization would be simpler than inner classes due to the absence of interaction with inheritance. On the other hand, it would be hard to model the implementation scheme of delegates, since it depends on Java's reflection features.

Other core calculi for Java. There have been proposed several calculi [5, 19, 17, 6] to study formal properties and extensions of Java; none of them, however, treats inner classes, although we don't see any inherent difficulty to integrate inner classes into their calculi.

8 Conclusions and Future Work

We have formalized two styles of semantics for inner classes: a direct style and a translation style, where semantics is given by compilation to a low-level language without inner classes, following Java's Inner Classes Specification. We have proved that the two styles correspond, in the sense that the translation commutes with the high-level reduction relation in the direct semantics. Besides deepening our own understanding of inner classes, this work has uncovered a significant underspecification in the official specification.

For future work, the interaction between inner classes and access restrictions in Java is clearly worth investigating. We also hope to be able to model Java's other forms of inner classes: anonymous classes and local classes, which can be declared in method bodies; these are slightly more complicated, since method arguments (not just fields) can occur in them as free variables, but we expect they can be captured by a variant of FJI.

Acknowledgments. This work was supported by the University of Pennsylvania and the National Science Foundation under grant CCR-9701826, *Principled Foundations for Programming with Objects*. Igarashi is a research fellow of the Japan Society of the Promotion of Science.

We would like to thank bug parade in Java Developer Connection (`http://developer.java.sun.com/developer/bugParade/index.html`) for providing useful information. Comments from the anonymous referees of POPL'99, FOOL7, and ECOOP2000 helped us improve the final presentation.

References

[1] Martín Abadi. Protection in programming-language translations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, pages 868–883. Springer-Verlag, July 1998. also appeared as DEC SRC Research Report 154 (April 1998).

[2] Anasua Bhowmik and William Pugh. A secure implementation of Java inner classes. Handout from PLDI '99 Poster Session. Available through `http://www.cs.umd.edu/~pugh/java`.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices volume 33 number 10, pages 183–200, Vancouver, BC, October 1998.

[4] Patrick Chan and Rosanna Lee. *The Java Class Libraries*, volume 2. Addison-Wesley, Reading, MA, second edition, October 1997.

[5] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java Type System Sound? *Theory and Practice of Object Systems*, 7(1):3–24, 1999. Preliminary version in ECOOP '97.

[6] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages (POPL)*, San Diego, January 1998. ACM.

[7] Neal Glew. Object closure conversion. In Andrew Gordon and Andrew Pitts, editors, *Proceedings of the 3rd International Workshop on Higher Order Operational Techniques in Semantics (HOOTS'99)*, volume 26 of *Electronic Notes in Theoretical Computer Science*, Paris, France, September 1999. Elsevier. Available through `http://www.elsevier.nl/locate/entcs/volume26.html`.

[8] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA, 1983.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.

[10] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Linda M. Northrop, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, volume 34, number 10, pages 132–146. ACM Press, October 1999.

[11] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. Technical Report MS-CIS-99-23, University of Pennsylvania, Philadelphia, PA, November 1999. Available through `http://web.yl.is.s.u-tokyo.ac.jp/~igarashi/papers.html`.

[12] JavaSoft. Inner classes specification, February 1997. Available through `http://java.sun.com/products/JDK/1.1/`.

[13] Ole Lehrmann Madsen. Semantic analysis of virtual classes and nested classes. In Linda M. Northrop, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, volume 34, number 10, pages 114–131, Denver, CO, October 1999. ACM Press.

[14] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1989.

[15] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, 1993.

[16] Microsoft. Microsoft Java SDK 3.2 documentation. Available online through `http://www.microsoft.com/Java/sdk/32/`, 1999.

[17] Tobias Nipkow and David von Oheimb. Java$_{light}$ is type-safe — definitely. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 161–170, San Diego, January 1998. ACM.

[18] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[19] Don Syme. Proving Java type soundness. Technical Report 427, Computer Laboratory, University of Cambridge, June 1997.

Syntax:

$$\mathtt{L} ::= \mathtt{class\ C} \triangleleft \mathtt{C}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}$$

$$\mathtt{K} ::= \mathtt{C}(\overline{\mathtt{C}}\ \overline{\mathtt{f}})\ \{\mathtt{super}(\overline{\mathtt{f}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}$$

$$\mathtt{M} ::= \mathtt{C\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\uparrow \mathtt{e};\}$$

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.m}(\overline{\mathtt{e}}) \mid \mathtt{new\ C}(\overline{\mathtt{e}})$$

Computation:

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{f}_i \longrightarrow \mathtt{e}_i}$$

$$\frac{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{e}_0)}{\begin{array}{c}(\mathtt{new\ C}(\overline{\mathtt{e}})).\mathtt{m}(\overline{\mathtt{d}})\\ \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ C}(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e}_0\end{array}}$$

Subtyping:

$$\mathtt{C} <: \mathtt{C} \qquad \frac{\mathtt{C} <: \mathtt{D} \qquad \mathtt{D} <: \mathtt{E}}{\mathtt{C} <: \mathtt{E}}$$

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \triangleleft \mathtt{D}\ \{\ldots\}}{\mathtt{C} <: \mathtt{D}}$$

Expression typing:

$$\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})$$

$$\frac{\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \qquad \mathit{fields}(\mathtt{C}_0) = \overline{\mathtt{C}}\ \overline{\mathtt{f}}}{\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i \in \mathtt{C}_i}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e}_0 \in \mathtt{C}_0 \qquad \mathit{mtype}(\mathtt{m},\mathtt{C}_0) = \overline{\mathtt{D}} \to \mathtt{C}\\ \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}\end{array}}{\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{C}}$$

$$\frac{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{f}} \qquad \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{C}} \qquad \overline{\mathtt{C}} <: \overline{\mathtt{D}}}{\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C}}$$

Method typing:

$$\frac{\begin{array}{c}\overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C} \vdash \mathtt{e}_0 \in \mathtt{E}_0 \qquad \mathtt{E}_0 <: \mathtt{C}_0\\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \triangleleft \mathtt{D}\ \{\ldots\}\\ \text{if } \mathit{mtype}(\mathtt{m},\mathtt{D}) = \overline{\mathtt{D}} \to \mathtt{D}_0,\\ \text{then } \overline{\mathtt{C}} = \overline{\mathtt{D}} \text{ and } \mathtt{C}_0 = \mathtt{D}_0\end{array}}{\mathtt{C}_0\ \mathtt{m}\ (\overline{\mathtt{C}}\ \overline{\mathtt{x}})\ \{\uparrow \mathtt{e}_0;\}\ \mathtt{OK\ IN\ C}}$$

Class typing:

$$\frac{\begin{array}{c}\mathtt{K} = \begin{array}{l}\mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{C}}\ \overline{\mathtt{f}})\\ \quad \{\mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\}\end{array}\\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C}\end{array}}{\mathtt{class\ C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}}$$

Field lookup:

$$\mathit{fields}(\mathtt{Object}) = \bullet$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathit{fields}(\mathtt{D}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}\end{array}}{\mathit{fields}(\mathtt{C}) = \overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{C}}\ \overline{\mathtt{f}}}$$

Method type lookup:

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\ \{\uparrow \mathtt{e};\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \overline{\mathtt{B}} \to \mathtt{B}}$$

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}\end{array}}{\mathit{mtype}(\mathtt{m},\mathtt{C}) = \mathit{mtype}(\mathtt{m},\mathtt{D})}$$

Method body lookup:

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\ \mathtt{B\ m}\ (\overline{\mathtt{B}}\ \overline{\mathtt{x}})\ \{\uparrow \mathtt{e};\} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = (\overline{\mathtt{x}},\mathtt{e})}$$

$$\frac{\begin{array}{c}\mathtt{m} \text{ is not defined in } \overline{\mathtt{M}}\\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C} \triangleleft \mathtt{D}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\end{array}}{\mathit{mbody}(\mathtt{m},\mathtt{C}) = \mathit{mbody}(\mathtt{m},\mathtt{D})}$$

Fig. 4. FJ Definitions

Jam - A Smooth Extension of Java with Mixins*

Davide Ancona, Giovanni Lagorio, and Elena Zucca

Dipartimento di Informatica e Scienze dell'Informazione
Via Dodecaneso, 35,16146 Genova (Italy)
{davide,lagorio,zucca}@disi.unige.it

Abstract. In this paper we present Jam, an extension of the Java language supporting *mixins*, that is, parametric heir classes. A mixin declaration in Jam is similar to a Java heir class declaration, except that it does not extend a fixed parent class, but simply specifies the set of fields and methods a generic parent should provide. In this way, the same mixin can be instantiated on many parent classes, producing different heirs, thus avoiding code duplication and largely improving modularity and reuse. Moreover, as happens for classes and interfaces, mixin names are reference types, and all the classes obtained instantiating the same mixin are considered subtypes of the corresponding type, hence can be handled in a uniform way through the common interface. This possibility allows a programming style where different ingredients are "mixed" together in defining a class; this paradigm is somehow similar to that based on multiple inheritance, but avoids the associated complications.
The language has been designed with the main objective in mind to obtain, rather than a new theoretical language, a working and smooth extension of Java. That means, on the design side, that we have faced the challenging problem of integrating the Java overall principles and complex type system with this new notion; on the implementation side, that we have developed a Jam to Java translator which makes Jam sources executable on every Java Virtual Machine.

1 Introduction

In the last years, the notion of *parametric heir class* or *mixin* (following the terminology originally introduced in [17,15]) has attracted great interest in the programming languages community. As the first name suggests, a mixin is a uniform extension of many different parent classes with the same set of fields and methods, that is, a class-to-class function. To be more concrete, let us consider a schematic class declaration in Java.

```
class H1 extends P1 { decs }
```

where P1 is some parent class and *decs* denotes a set of field and method declarations. In Java, as in most other object-oriented programming languages, if we want to extend another parent class, say P2, with *the same* set of fields and methods, then we have to write a new independent declaration, duplicating *decs*.

[*] Partially supported by Murst - Tecniche formali per la specifica, l'analisi, la verifica, la sintesi e la trasformazione di sistemi software

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 154–178, 2000.
© Springer-Verlag Berlin Heidelberg 2000

```
class H2 extends P2 { decs }
```

Consider now a language allowing to give a name, say `M`, to *`decs`*, and instantiating `M` on different parent classes, e.g., `P1` and `P2`, obtaining different heir classes equivalent to `H1` and `H2` above.

```
mixin M { decs }
class H1 = M extends P1 ;
class H2 = M extends P2 ;
```

Then we say that `M` is a *mixin.*

A mixin declaration resembles a usual heir class declaration, except for the fact that a mixin does not refer to a fixed parent class, but simply specifies the set of fields and methods a parent should provide. The fact that the same mixin can be instantiated on many parent classes avoids code duplication and largely improves modularity and reuse. The name refers to the fact that in a language supporting mixins it is possible to "mix", in some sense, different ingredients during class creation, as nicely illustrated through the jigsaw puzzle metaphor in [5]. This paradigm is similar to multiple inheritance, but avoids the associated complications.

Mixin-based programming has been now extensively studied both on the methodological and foundational point of view [6,5,2,3]. The results can be summarized as follows. First, the mixin notion is not strictly related to object-oriented programming but can be formulated in general in the context of module composition (a *mixin module* is a module where some components are not defined but expected to be provided by some other module). This notion allows to have a clean and unifying view of different linguistic mechanisms for composing modules. Then, the intuitive understanding of a mixin as a class-to-class function (or, in the general case, module-to-module function) can be actually supported by a rigorous mathematical model [2,3].

Despite of this advanced state of the art, few attempts have been tried to designing real programming languages supporting mixins. As already mentioned, the first use of the word mixin as a technical term originates with the LISP community [15,19]. After that, at our knowledge, there exist only a proposal for extending ML [12], a working extension of Smalltalk [7] and a proposal for a Java-like mixin language [13] (whose relation with our work will be discussed in detail in Sect.6).

In this paper, we present Jam[1], a working and smooth extension of Java with mixins. By these two adjectives we mean that our main aim is to produce an executable and minimal extension of Java, rather than define a new theoretical language supporting mixins. More precisely, Jam is an upward-compatible extension of Java 1.0 (apart from two new keywords), a great effort has been spent in integrating mixin-related features with the Java overall design principles, the type system is a natural extension of the Java type system with a new kind of types (mixin types), the dynamic semantics is directly defined by translation into Java and, finally, this translation has been implemented by a Jam to

[1] **Ja**va + **m**ixin = **Jam**

Java translator which makes Jam immediately executable on every Java Virtual Machine.

The structure of the presentation is as follows. In Sect. 2 we provide an introduction to Jam, through some examples, and illustrate and motivate in detail our design choices. In Sect. 3 we outline the abstract syntax and the formal static semantics (the full definition can be found in [1]). In Sect. 4 we define a translation from Jam into Java and state the correctness of this translation w.r.t. static semantics (that is, correct Jam programs are expanded into correct Java programs; this also ensures the soundness of the Jam type system). In Sect. 5 we briefly describe the implementation. Finally in Sect. 6 we provide a detailed comparison with related work and outline further research directions.

An extended version of this paper, including the full type system, the proof of correctness of the translation and more examples and discussions, is [1].

The Jam compiler and its sources are available at the following URL: `http://www.disi.unige.it/person/LagorioG/jam`.

2 Introduction and Rationale

2.1 An Example

Fig. 1 shows the declaration of the mixin `Undo`. We use `typewriter` style for code fragments. This mixin, as the name suggests, provides an "undo" mechanism that supports restoring of the text before the latest modification.

```
mixin Undo {
 inherited String getText() ;
 inherited void setText(String s) ;
 String lastText;
 void setText (String s) { lastText = getText() ; super.setText(s) ; }
 void undo () { setText(lastText); }
}
```

Fig. 1. Mixin declaration

As shown in the example, a mixin declaration is logically split in two parts: the declarations of the components which are expected to be provided by the parent class, prefixed by the `inherited` modifier, and the declarations of the components defined in the mixin. Note that defined components can override/hide inherited components, as it happens for usual heir classes.

The mixin `Undo` can be instantiated on classes that define two non-abstract methods `getText` and `setText`, with types as specified in the `inherited` declaration.

Fig. 2 shows an example of instantiation; we have used as parent a class `Textbox` which extends a generic class `Component`. In the instantiation no constructors are specified for the new class `TextboxWithUndo` (they should be declared between the curly braces) and so, as in Java, it is assumed that the class has

```
class Textbox extends Component {
 String text ;
 ...
 String getText() { ... }
 void setText(String s) { ... }
}
class TextboxWithUndo = Undo extends Textbox {}
```

Fig. 2. Mixin instantiation

only the default constructor. To obtain a correct instantiation `Textbox` must define the mixin `inherited` part by implementing the methods `getText` and `setText`. These methods must have the same return and arguments type and equivalent (substitutable)[2] `throws` clause w.r.t. the corresponding `inherited` declaration. The classes obtained by instantiating the mixin provide, in addition to the methods `getText` (inherited from parent class) and `setText` (inherited and overridden), all other fields and methods of the class `Textbox`, the method `undo` and the field `lastText`.

The expected semantics of mixin instantiation can be informally expressed by the following *copy principle*:

> A class obtained instantiating a mixin M on a parent class P should have the same behavior as a usual heir of P whose body contains a copy of all the components defined in M.

This principle corresponds to the "natural" semantics for mixin instantiation, that is, the semantics one would obtain defining a "hand-made" subclass. A class implementing the mixin `inherited` part can nevertheless be an invalid parent for instantiation, since there is another requirement to be met: the heir class obtained instantiating the mixin must be a correct Java heir class. This leads to a set of constraints which are described in detail in Sect. 2.3.

What we have seen so far demonstrates the use of a mixin declaration as a *scheme*, that is, a parametric heir class that can be instantiated on different classes. In this way we avoid code duplication, a good result in itself, but Jam allows something more: a mixin can be used as a *type* and a mixin instance[3] is a subtype of both the mixin and the parent class on which it has been instantiated.

This allows the programmer to manipulate objects of any mixin instance by using the common interface specified by the mixin declaration (see Fig. 3).

An important consequence is that Jam supports a programming style (sometimes called mixin-based [6]) where different ingredients are "mixed" together in defining a class. This paradigm has been advocated [5] on the methodological

[2] That is, every exception declared in one clause must be a subtype of an exception declared in the other, and conversely.

[3] We will call *mixin instance* a class obtained by instantiating a mixin, to be not confused with an *instance* of a class.

```
class TextboxWithUndo = Undo extends Textbox {}
class BreakIteratorWithUndo = Undo extends java.text.BreakIterator {}
class TestUndo {
 void f() {
  g( new TextboxWithUndo() ) ;
  g( new BreakIteratorWithUndo() ) ;
 }
 void g(Undo u) {
  u.setText("foo") ; u.setText("bar") ;
  System.out.println("Previous text: "+u.lastText) ;
  System.out.println("Current text : "+u.getText());
 }
}
```

Fig. 3. Use of mixin types

side since it allows to recover some of the expressive power of multiple inheritance without introducing its complication; however the novelty of Jam is that mixin-based programming is rigorously introduced in the context of a strongly typed language.

2.2 Other Components of a Mixin Declaration

In the simple example presented in the previous section we have not included *all* the kinds of components which can appear in a mixin declaration.

Indeed, following the design principle that a mixin should be as similar as possible to a usual heir class, mixins should provide all their features. In the sequel we illustrate each of them in detail highlighting and justifying some restrictions.

Interfaces. A mixin can implement many interfaces in exactly the same way a class does.

Constructors. In Jam constructors cannote be declared in a mixin, but only for each single mixin instance at the point of instantiation. From a technical point of view, it would be conceivable to declare constructors in mixins, handling them as components which are not part of the mixin type, as we do for static components (see below). However, we have preferred this choice since from a methodological point of view constructors are tightly tied-up with the implementation of their own class, so their signatures tend to be not very general.

Inherited instance fields. In a mixin it is possible to access inherited (instance) fields in the same way as a usual heir class does: using the field name `id` or the forms `this.id` and `super.id` (the latter is needed when a defined field hides an inherited one).

Static members. Although in Jam static components are declared in the same way as instance components except, of course, the use of the `static` modifier, their visibility is different: they are not considered part of the mixin type. Consider, for example, the following code fragment:

```
mixin M { static void m() {} static int f ; }
```

We do not allow in Jam invocations `M.m()` or *e*`.m()` with *e* of type `M`. However, for each class `H` obtained instantiating `M`, invocations `H.m()` or *e*`.m()` with *e* of type `H` are legal. The same rule holds for fields[4]. In other words, every class that is an instance of `M` has "its own copy" of static components declared in the mixin. Other choices are technically possible:

- sharing only one copy of the static components declared in the mixin between all mixin instances; in this case accessing static members through the mixin type should be allowed as well;
- leave to the programmer (introducing a new keyword, or analogous mechanisms) the decision whether a component should be shared between all the mixin instances or not.

In Jam, we have chosen the "unshared" version because, in this way, a mixin instantiation on a parent class is equivalent to that obtained by copying the mixin body in the declaration of the new class, as requested by the copy principle. Static components can be inherited (of course, they are not part of mixin type either) but, like in Java, static methods cannot be abstract.

2.3 Constraints on Instantiation

As mentioned in Sect. 1, the fact that a class `P` provides an implementation for the `inherited` part of a mixin `M` is not enough to ensure that `P` can be correctly used as a parent for `M`. Indeed, in addition to methods declared `inherited` in `M`, the class `P` can contain some *other* methods which could interfere, in various ways, with methods in `M`. Let us briefly illustrate the different interference cases.

Illegal overriding/hiding A method in `P` is illegally overridden (hidden) (see 8.4.6.3 in [14]) by a defined method in `M` if it has the same signature (name and arguments type) but either different return type, or different kind (`instance` or `static`) or incompatible `throws` clause. This is not correct in Jam as well as in Java.

Unexpected overriding/hiding A method in `P` is incidentally overridden (hidden) by a method defined in `M` if it has the same name, arguments type, return type, kind and a compatible (see 8.4.4 in [14]) `throws` clause. For instance, instantiating the mixin `Undo` on a class with a `void undo()` method produces an unexpected overriding. This situation looks somehow undesirable, since there is some overriding which was not planned when declaring the mixin; however, our choice for Jam has been to consider these instantiations to be legal, leaving to the programmer the responsibility of avoiding them when the additional overriding is undesired[5]. Indeed, different choices would sensibly complicate either

[4] We maintain this alternative syntax for compatibility reasons only, see 15.10.1 of [14].

[5] A Jam compiler may issue a warning in such cases.

the static (if the choice is to forbid) or dynamic (if the choice is to keep both versions) semantics, while ours is the natural extension to mixins of usual heir classes semantics. See Sect.6 for some further discussion on this point.

Ambiguous overloading There exist contexts in which the presence of the method in `P` makes ambiguous, w.r.t. overloading resolution, an invocation of the method in `M`. Let us clarify this case with an example. Assume that the method `Undo.undo` contains the call `setText(null)`; this invocation is statically correct. Suppose now to instantiate `Undo` on a class `Boom` which defines, besides the methods `String getText()` and `void setText(String)`, also another method `void setText(Integer)`. In this case the call `setText(null)` becomes ambiguous. Indeed, `null` can be implicitly converted to any reference type, hence both methods are applicable and neither is more specific (see 15.11.2 in [14]).

In general, if two methods have the same name, then the addition of one may make ambiguous, w.r.t. overloading resolution, an invocation of the other if and only if they have the same number and type of arguments except for some argument for which they have two different reference types. Alternatively we could have defined less strict rules by forbidding the instantiation only when some method body in the mixin contains a method invocation that would become ambiguous (as in the example).

2.4 Overloading

The Java rules for overloading resolution (see 15.11.2 in [14]) smoothly extend to Jam, just including mixin types among other reference types and taking into account in the definition of "more specific" the fact that every mixin instance is a subtype of (hence, can be converted to) the corresponding mixin type. However, some special care is needed for handling the situation when there is an overloading conflict between an inherited and a defined method in a mixin. We illustrate the problem in terms of the following example.

```
class A {}
class B extends A {}

class Parent {
 void f(B b) {}
}

class Heir extends Parent {
 void f(A a) {}
}
```

```
mixin M {
 inherited void f(B b) ;
 void f(A a) {}
}

class Test {
 void test(Heir h, B b, M m) {
  h.f(b) ; // ambiguous
  m.f(b) ; // ambiguous?
 }
}
```

Fig. 4. Overloading conflict between inherited and defined methods

In the first part of the code shown in Fig.4, B is a subtype of A and Heir is a subtype of Parent. The class Parent defines a method named f with one argument of type B, while its subclass Heir defines a another method with the same name and argument's type A. Due to the symmetry of the situation, the invocation h.f(b), where h and b are of type Heir and B, respectively, is ambiguous, since there are two applicable methods and neither is more specific (see 15.11.2.2 in [14]).

If we now consider the declaration of the mixin M, the situation is exactly analogous to the preceding: a (parametric) heir class defines a method whose argument type is a supertype of the argument type of a method with the same name in the parent class. Hence, we expect the invocation of m.f(b), where m has type M, to be ambiguous as well.

In order to achieve this goal, we assume that inherited methods in a mixin M are annotated with a type (that is, considered to be declared within the corresponding module; see [1] for the precise formal definition of annotations) which is not M but a special type *Parent*(M) which represents the generic parent on which the mixin can be instantiated, and it is assumed to be a supertype of M.

2.5 Use of this in Mixins

A last delicate point in the Jam type system concerns the use of the keyword this, which denotes, in an instance method (resp. constructor), the current object on which the method has been invoked (the current object to be constructed). In a method or constructor declared in a class C, the expression this has static (compile-time) type C in Java (see 15.7.2. in [14]). Now, we have to decide which should be the static type of this in a method defined in a mixin M. Since we want to be able to type-check the mixin declaration independently from its (possibly future) instantiations, the only possibility is to assume that this has static type M, since this is the only type available at mixin declaration's time. However, this is in conflict with the fact that we expect that in a class H instance of a mixin M the expression this has static type H, as it happens for usual heir classes. More precisely, having correctly type-checked the mixin declaration under the assumption that this has type M does not guarantee that (the Java class H corresponding to) a mixin instance (following the copy principle) is always a correct Java class, since in Java this has type H in this class. This can lead to unsound situations in some subtle cases involving overloading. Let us consider the example in Fig.5.

The class A declares two methods named f with argument's type a mixin M and an instance H of M, respectively. In the invocation of f inside the method g declared in M, since this has type M, the expression A.f(this) has type int, hence can be correctly assigned to the variable i.

Now, if the expected semantics of H, following the copy principle, is to be equivalent to the class shown in the figure where the declaration of g has been copied into the body, then the invocation A.f(this) has now type boolean, hence cannot be used for initializing the variable i.

This is a particular case of a more general problem: in a mixin declaration, in Jam, there is no way to refer to the parametric types of either the parent or

```
class A {
 static int f(M m) { ... }
 static boolean f(H h) { ... }
}

mixin M {
 void g() {
  int i = A.f(this) ;
 }
}
```

```
class H = M extends Object {}

// ``Equivalent'' declaration for H
class H ... {
 void g() {
  int i = A.f(this) ; // Boom !!!
 }
}
```

Fig. 5. Problem in using `this` in mixins

the heir class resulting from the instantiation. See the following section for more comments about that.

In order to avoid these situations, we have taken for Jam a quite drastic design decision, that is, to forbid the use of `this` as argument in method and constructor invocations inside a mixin.

2.6 Limitations

The main limitation of the language is that in a mixin it is not possible to refer either to the "generic" parent class to which the mixin will be applied or to the "generic" heir class obtained by instantiation. As an example, let us consider the following declarations of a parent class `P` and an heir class `H`.

```
class P {
 static int counter ;
}
class H extends P {
 static int counter ;
 static void incrThat() { ++P.counter ; }
 ...
 int value ;
 public boolean equals(Object that) {
  if (that instanceof H) return ((H)that).value == value ;
  return false ;
 }
}
```

The definition of `H` cannot be "abstracted" in a mixin definition, for two reasons.

- The method `incrThat` explicitly refers to the parent class `P` since the static field `counter` of the superclass has been hidden by a declaration in `H` (this could be achieved by extending the use of the keyword `super` to such cases).
- The method `equals` uses the name of the heir class `H` which is unknown for a mixin (note that this cannot be achieved using `M`, since that would be kept as `M` in all mixin instances following the copy principle).

More generally, a class H heir of P cannot be “abstracted” into a mixin when it contains:

- (explicit) references to types H and P;
- invocations of H/P constructors.

If H/P are only used for accessing static members, then H can be “abstracted” except for some cases involving hiding (as shown by the example). In Sect. 6 we discuss possible solutions to this problem.

3 Outline of The Formal Definition

In this section we outline the abstract syntax and the static semantics of Jam; the full definition can be found in [1].

The implemented version of Jam is an upward-compatible extension of Java 1.0 (apart the fact that mixin and inherited are keywords in Jam); an extension to Java 1.1 would require some more work at the implementative level, but seems in principle not to introduce new problems. Indeed the main enhancement in Java 1.1 is the introduction of inner classes, whose semantics is specified by flattening them to usual top-level classes.

The formal definition in [1] only considers a subset of the language chosen to be a minimal but sufficient set for our aim, which is to analyze how the Java type system must be enriched in order to support mixin types (the soundness of this extension will be stated in the next section). Excluded features fall under two main categories: those which are orthogonal with our aim, like multithreading and inner classes and those whose semantics can be trivially derived, like the for loop. In particular, we have excluded the following features: arrays, final and access modifiers, features related with linking native code and multithreading. We have included the following features not considered in [10]: constructors, static members, checked exceptions[6], abstract classes and methods, method invocations and field accesses via super.

Notations. We use the typewriter style for terminal and *italic* for non terminal symbols. The terminals iname, cname and mname indicate interface, class and mixin names respectively. A generic name is indicated by name. We use the following notations:

- A^* to indicate a sequence of zero or more occurrences of A,
- A^+ to indicate a sequence of one or more occurrences of A,
- $[A]$ to indicate that A is optional,
- $A^{\circledast}$ to indicate a set of occurrences of A, that is, a sequence in which there are no repetitions and the order is immaterial,
- $A^{\oplus}$ to indicate a non empty set of occurrences of A.

Fig. 6 contains the part of the Jam abstract syntax related to interface, class and mixin declarations (the Jam specific productions are the first three only).

[6] Checked exceptions have been considered in a recent improved version [11].

$$
\begin{array}{rcl}
\mathit{ref\text{-}type} & ::= & \mathtt{mname} \\
\mathit{cdecl} & ::= & [\ \mathtt{abstract}\ ]\ \mathtt{class}\ \mathtt{cname}\ =\ \mathtt{mname}\ \mathtt{extends} \\
 & & \mathtt{cname}\ \{\mathit{constructor}^{\circledast}\} \\
\mathit{mdecl} & ::= & \mathtt{mixin}\ \mathtt{name}\ \mathtt{implements}\ \mathtt{iname}^{\circledast} \\
 & & \{\ \langle\ [\mathtt{inherited}]\ \mathit{field}\ \rangle^{\circledast}\ \langle\ [\mathtt{inherited}]\ \mathit{cmeth}\ \rangle^{\circledast}\ \} \\
 & & \\
\mathit{prog} & ::= & \mathit{decl}^{\circledast} \\
\mathit{decl} & ::= & \mathit{idecl} \mid \mathit{cdecl} \mid \mathit{mdecl} \\
\mathit{simple\text{-}type} & ::= & \mathit{prim\text{-}type} \mid \mathit{ref\text{-}type} \\
\mathit{ref\text{-}type} & ::= & \mathtt{iname} \mid \mathtt{cname} \\
\mathit{prim\text{-}type} & ::= & \mathtt{int} \mid \mathtt{boolean} \\
\mathit{ret\text{-}type} & ::= & \mathit{simple\text{-}type} \mid \mathtt{void} \\
\mathit{exc\text{-}type} & ::= & \mathtt{cname}^{\circledast} \\
\mathit{cdecl} & ::= & [\ \mathtt{abstract}\ ]\ \mathtt{class}\ \mathtt{cname}\ \mathtt{extends}\ \mathtt{cname} \\
 & & \mathtt{implements}\ \mathtt{iname}^{\circledast}\ \{\ \mathit{constructor}^{\circledast}\ \mathit{field}^{\circledast}\ \mathit{cmeth}^{\circledast}\ \} \\
\mathit{idecl} & ::= & \mathtt{interface}\ \mathtt{iname}\ \mathtt{extends}\ \mathtt{iname}^{\circledast}\ \{\ \mathit{imeth}^{\circledast}\ \} \\
\mathit{imeth} & ::= & \mathtt{abstract}\ \mathit{ret\text{-}type}\ \mathtt{name}\ \mathit{params}\ \mathtt{throws}\ \mathit{exc\text{-}type}\ ; \\
\mathit{params} & ::= & (\langle\ \mathit{simple\text{-}type}\ \mathtt{name}\ \rangle^{*}) \\
\mathit{constructor} & ::= & \mathtt{cname}\ \mathit{params}\ \mathtt{throws}\ \mathit{exc\text{-}type}\ \{\ \mathtt{super}(\mathit{expr}^{*})\ ;\ \mathit{stmts}\ \} \\
\mathit{cmeth} & ::= & [\ \mathtt{static}\ ]\ \mathit{ret\text{-}type}\ \mathtt{name}\ \mathit{params}\ \mathtt{throws}\ \mathit{exc\text{-}type}\ \mathit{mbody} \mid \\
 & & \mathit{imeth}
\end{array}
$$

Fig. 6. Jam abstract syntax

In Jam an alternative way to define a (possibly `abstract`) class is to instantiate a mixin on an existing class, specifying the constructors of the new class.

A mixin declaration logically consists of two parts: the former contains the declarations of the defined components, while the latter contains the inherited components declarations, that is, the declarations of the components that should be provided by the parent class on which the mixin will be instantiated. These components are labelled with the `inherited` modifier. Moreover, the set of the implemented interfaces is specified.

In Fig. 7 we define the Jam types. A generic *type* can be a reference type, a primitive type (both defined in Fig. 6) or `nil` (the type of `null`). A *field type* consists of a simple type and a (field) *kind* indicating whether the field is instance or static. The *arguments type* (of a method or constructor) is a sequence, possibly empty, of simple types. A *constructor type* consists of the arguments type and the set of declared exceptions (the type *exc-type* is defined in Fig. 6). A *method type* consists of the kind, the return type and the set of declared exceptions. A *fields type* is a set of fields, that is, pairs consisting of a field name and a field type. A fields type is *legal* if field names are distinct. Analogously, a *methods type* is a set of methods, that is, pairs consisting of a *signature* (a method name qualified by the types of the arguments) and a method type; it is *legal* when all signatures are distinct. In the following we will consider only legal fields and methods types. The *constructors type* is a non-empty set of constructor types. Note that every class has always at least one constructor (if it is not explicitly given the default one is assumed).

$$
\begin{aligned}
\mathit{type} &::= \mathit{ref\text{-}type} \mid \mathit{prim\text{-}type} \mid \mathtt{nil}\\
\mathit{field\text{-}type} &::= \mathit{field\text{-}kind}\ \mathit{simple\text{-}type}\\
\mathit{field\text{-}kind} &::= \mathtt{instance} \mid \mathtt{static}\\
\mathit{args\text{-}type} &::= \mathit{simple\text{-}type}^{*}\\
\mathit{constr\text{-}type} &::= \mathit{args\text{-}type}\ \mathtt{throws}\ \mathit{exc\text{-}type}\\
\mathit{meth\text{-}type} &::= \mathit{meth\text{-}kind}\ \mathit{ret\text{-}type}\ \mathtt{throws}\ \mathit{exc\text{-}type}\\
\mathit{meth\text{-}sig} &::= \mathtt{name}, \mathit{args\text{-}type}\\
\mathit{meth\text{-}kind} &::= \mathtt{instance} \mid \mathtt{abstract} \mid \mathtt{static}\\
\mathit{fields\text{-}type} &::= \langle \mathtt{name} : \mathit{field\text{-}type} \rangle^{\circledast}\\
\mathit{meths\text{-}type} &::= \langle \mathit{meth\text{-}sig} : \mathit{meth\text{-}type} \rangle^{\circledast}\\
\mathit{constrs\text{-}type} &::= \mathit{constr\text{-}type}^{\oplus}\\
\\
\mathit{module\text{-}type} &::= \mathit{fields\text{-}type}\ \mathit{meths\text{-}type}\\
\mathit{class\text{-}type} &::= \mathit{class\text{-}kind}\ \mathit{constrs\text{-}type}\ \mathit{module\text{-}type}\\
\mathit{class\text{-}kind} &::= \mathtt{abstract} \mid \mathtt{concrete}\\
\mathit{interface\text{-}type} &::= \mathit{meths\text{-}type}\\
\mathit{mixin\text{-}type} &::= \mathit{module\text{-}type}\ \mathtt{inherited}\ \mathit{module\text{-}type}
\end{aligned}
$$

Fig. 7. Jam types

A *module type* consists of a set of fields and a set of methods. A *class type* consists of a module type, a kind and a set of constructors. An *interface type* consists of a set of methods (in our subset we do not consider the `final` modifier, hence an interface cannot have fields). Finally, a *mixin type* consists of two module types: the defined type and the inherited type, that is the expected parent type.

A Jam program contains both type information and information needed at runtime (that is, the method bodies). To simplify the formal definition, following the approach used in [11], we consider two components that can be extracted in a trivial way from a program: the environment Γ, that contains the type information, and the remaining part of program consisting in a set of *body declarations*, that is, constructor and method bodies of classes and mixins (fields information is contained in Γ). The syntax of these two components is given in Fig. 8.

$$
\begin{aligned}
\mathit{env} &::= \mathit{basic\text{-}type\text{-}assertion}^{\circledast}\\
\mathit{basic\text{-}type\text{-}assertion} &::= \mathtt{cname}\ \mathrm{is}_c\ \mathit{class\text{-}type} \mid \mathtt{cname}\ <_c^1\ \mathtt{cname} \mid \mathtt{name}\ \lhd_i^1\ \mathtt{iname} \mid\\
&\ \mathtt{iname}\ \mathrm{is}_i\ \mathit{interface\text{-}type} \mid \mathtt{iname}\ <_i^1\ \mathtt{iname} \mid\\
&\ \mathtt{mname}\ \mathrm{is}_m\ \mathit{mixin\text{-}type} \mid \mathtt{cname}\ \lhd_m\ \mathtt{mname}\\
\mathit{body\text{-}decl} &::= \mathtt{class}\ \mathtt{cname}\ \{\ \mathit{constructor}^{\circledast}\ \mathit{cmeth}^{\circledast}\ \} \mid \mathtt{mixin}\ \mathtt{name}\ \{\ \mathit{cmeth}^{\circledast}\ \}
\end{aligned}
$$

Fig. 8. Environments and body declarations

We assume that in the environment extraction process a check is performed for avoiding duplicate declarations. Hence, the static correctness of a Jam program can be expressed by the validity of the two following judgments:

$\Gamma \vdash \diamond$
$\Gamma \vdash \{BD_1, \ldots, BD_n\} \diamond_{\text{Prog}}$
The former means that Γ is a well-formed environment so that, for instance, the subclass relationship is acyclic; the latter indicates that all the body declarations are well-formed w.r.t. the type information in Γ. The validity of these two judgments is defined inductively introducing other judgments relative to subcomponents. Because of lack of space, here we omit all the typing rules defininining the validity of the second judgment and only show some significant typing rules defining the validity of the first judgment (well-formedness of environments), in particular all those Jam-specific; for the missing metarules we provide a brief informal explanation. The full type system can be found in [1].

An environment is a set of basic type assertions having the following informal meaning:

- C is$_c$ K KST FST MST : the class C of kind K declares the specified constructors (KST), fields (FST) and methods (MST)
- $C <^1_c C'$: the class C directly extends the class C'
- $T \lhd^1_i I$: the module (either class or mixin) T directly implements the interface I
- I is$_i$ MST : the interface I declares the methods specified in MST
- $I <^1_i I'$: the interface I directly extends the interface I'
- M is$_m$ $MODT$ `inherited` $MODT'$: the mixin M declares the defined components $MODT$ and the inherited components $MODT'$
- $C \lhd_m M$: the class C has been defined instantiating the mixin M

Judgments of the metarules related to well-formedness of environments have the form $\Gamma \vdash \gamma$, with Γ an environment and γ a *type assertion*.

A group of metarules of the Jam type system defines relevant relations between reference types (that is, either classes, or interfaces, or mixins) which can be derived from the basic relations contained in the environment. In particular, the reflexive (on existing class types) and transitive closure of the relation $<^1_c$ is the *subclass* relation $\leq_c$; analogously the reflexive (on existing interface types) and transitive closure of $<^1_i$ is the *subinterface* relation $\leq_i$. The *implementation* relation from classes to interfaces is derived from $\lhd^1_i$ and the subclass and subinterface relations. The new relation introduced in Jam w.r.t. Java is that of *instantiation*, denoted $\lhd_m$, from a mixin instance to the corresponding mixin type. Finally, from all these relations we can derive a more general relation of *widening* (subtyping) between reference types, denoted by $\leq$. In Fig. 9 we show the metarule stating that a mixin instance is a subtype of the corresponding mixin type and that expressing reflexivity of widening for mixin types.

Another group of metarules defines subtyping relations for exceptions, fields, methods and module types. These relations basically express that a module type $MODT$ is a subtype of another module type $MODT'$ (written $MODT \leq_{mod} MODT'$) if it has more fields and/or methods; the common fields and methods must have *exactly* the same type, modulo equivalence of exceptions types $=_e$, defined by $\Gamma \vdash ET =_e ET'$ iff $\Gamma \vdash ET \leq_e ET'$ and $\Gamma \vdash ET' \leq_e ET$. Subtyping relation for exceptions types is defined in Fig. 9; note that it is possible that $E_i = E_j$ or $E'_i = E'_j$ holds for some i, j.

$$\frac{\Gamma \ \vdash \ C \lhd_m M}{\Gamma \vdash C \leq M} \qquad \frac{\Gamma \ \vdash \ M \ \text{is}_m \ MXT}{\Gamma \vdash M \leq M}$$

$$\frac{\begin{array}{l} \Gamma \vdash E_1 \leq_c E'_1 \ldots \ \Gamma \vdash E_n \leq_c E'_n \\ \Gamma \vdash \{E_1, \ldots, E_n\} \diamond_{\text{ExcType}} \ \ \Gamma \vdash \{E'_1, \ldots, E'_n\} \diamond_{\text{ExcType}} \end{array}}{\Gamma \vdash \{E_1, \ldots, E_n\} \leq_e \{E'_1, \ldots, E'_n\}} \qquad n \geq 0$$

Fig. 9. Relations on types

Other metarules define *type assignments*, that is, the fact that some Jam module (either class or interface or mixin) has a given type. In Fig. 10 we show the two metarules related to the two Jam-specific cases, that is a mixin and a class which is a mixin instance.

Set $MXT = FST\ MST\ \texttt{inherited}\ FST'\ MST'$

$$\frac{\begin{array}{l} \Gamma \vdash M \ \text{is}_m \ MXT \quad \Gamma \vdash MXT \diamond_{\text{MixinType}} \\ \Gamma \vdash I_1 : \emptyset \ MST_1 \ldots \Gamma \vdash I_n : \emptyset \ MST_n \end{array}}{\begin{array}{l} \Gamma \vdash M : \ FST'[FST] \\ \qquad (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n \overset{\Gamma}{\oplus} MST')[MST]_{\Gamma} \end{array}} \qquad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \{I \mid M \lhd^1_i \ I \in \Gamma\} \end{array}$$

Set:

$FST_c = FST'[FST_d], MST_c = (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n \overset{\Gamma}{\oplus} MST')[MST_d]_{\Gamma}$

$MST_m = (MST_1 \overset{\Gamma}{\oplus} \ldots \overset{\Gamma}{\oplus} MST_n \overset{\Gamma}{\oplus} MST_i)[MST_d]_{\Gamma}, CT = K\ KST\ \emptyset\ \emptyset$

$$\frac{\begin{array}{l} \Gamma \vdash C \ \text{is}_c \ CT \quad \Gamma \vdash CT \diamond_{\text{ClassType}} \\ \Gamma \vdash C \lhd_m M \\ \Gamma \vdash M \ \text{is}_m \ FST_d \ MST_d \ \texttt{inherited} \ FST_i \ MST_i \\ \Gamma \vdash C <^1_c C' \quad \Gamma \vdash C' : FST' \ MST' \\ \Gamma \vdash FST' \ MST' \leq_{mod} FST_i \ MST_i \\ \Gamma \vdash I_1 : MST_1 \ldots \Gamma \vdash I_n : MST_n \end{array}}{\Gamma \vdash C : FST_c \ MST_c} \qquad \begin{array}{l} n \geq 0 \\ \{I_1, \ldots, I_n\} = \{I \mid M \lhd^1_i \ I \in \Gamma\} \\ K = \texttt{concrete} \Rightarrow \\ \qquad \mathit{Kind}(MST_c) = \texttt{concrete} \\ \neg \mathit{MayBeAmbig}(MST', MST_m) \end{array}$$

Fig. 10. Type assignments

The type of a class, interface or mixin is a module type, that is, a pair consisting of a fields type and a methods type. The first metarule in Fig. 10 defines the type of a mixin. The fields type consists of the inherited fields type, updated by the defined fields. The methods type consists of the sum of the methods types of the implemented interfaces and of the inherited methods, updated by the defined methods. The auxiliary *update* operations on (legal) Jam fields and methods types, written _[_], return a new fields (resp. methods) type obtained updating the first argument with new fields (resp. methods), if this is possible, accordingly with Java rules on hiding, overloading and overriding. For instance, updating a methods type is undefined if we try to override a method with another which has the same signature and different return type. On methods types we define moreover a sum operation $\overset{\Gamma}{\oplus}$ which is basically set union, a part that, in the case of two methods with the same signature the operation is defined only if they have the same return type and either they are both `abstract` or one is `abstract` and the other is an instance method with compatible `throws` clause. In this case, the sum contains just one such method whose exceptions type is the "intersection" of the exceptions types, defined by

$E \in ET \overset{\Gamma}{\otimes} ET'$ iff either $E \in ET$ and $\exists E' \in ET'$ *s.t.* $\Gamma \vdash E \leq_c E'$ or conversely.

This operation is needed in the case a class inherits (from the parent class and implemented interfaces) many methods which differ only for the `throws` clauses.

For the formal definition of update and sum operations see [1].

The second metarule in Fig. 10 defines the type of a mixin instance. The fields type consists of the fields type of the superclass updated by the fields defined in the mixin. The methods type consists of the sum of the methods types of the implemented interfaces and the methods of the superclass, updated by the methods defined in the mixin. The predicate *MayBeAmbig* (omitted here) holds whenever the two arguments types may cause ambiguity, and thus make two methods incompatible in mixin instantiation as explained in Sect. 2.3.

Other analogous metarules define the type of an interface and of a standard Java class. We assume that all these metarules can be instantiated only when update operations are defined.

In Fig. 11 we show some of the metarules which define when a Jam environment is well-formed. More precisely, the judgment $\Gamma \vdash \Gamma'\diamond$ denotes that the declarations in Γ' are well-formed in the larger environment Γ. Indeed, we follow the approach in [10] of considering larger environment in order to correctly deal with mutual recursion between declarations. An environment Γ is well-formed if $\Gamma \vdash \Gamma\diamond$; in this case we also use the abbreviation $\Gamma \vdash \diamond$.

We show, in particular, the metarules which define well-formedness of mixin declarations and mixin instantiations; analogous rules hold for interface and usual class declarations.

$$\frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash M : FST\ MST}{\Gamma \vdash \Gamma' \cup \{M \ \text{is}_m\ MXT, M \lhd_i^1 I_1, \ldots , M \lhd_i^1 I_n\}\diamond} \qquad \Gamma'(M) = \bot$$

$$\frac{\Gamma \vdash \Gamma' \diamond \quad \Gamma \vdash C \ : \ FST\ MST}{\begin{array}{l}\Gamma \vdash \Gamma' \cup \{C \ \text{is}_c\ K\ KST\ \emptyset\ \emptyset, C <_c^1 C', C \lhd_m M,\\ \quad C \lhd_i^1 I_1, \ldots , C \lhd_i^1 I_n\}\diamond\end{array}} \qquad \begin{array}{l}\Gamma'(C) = \bot\\ \Gamma \not\vdash C' \leq_c C\end{array}$$

Fig. 11. Well-formed declarations

4 Jam to Java Translation

In this section, we give a formal definition of the dynamic semantics of Jam directly by translation into Java. The same approach of defining a Java extension by translation into Java has been taken for Pizza [18], a superset of Java which incorporates parametric polymorphism, higher-order functions and algebraic data types, and its recent evolution GJ (for "Generic Java") [9].

We first illustrate informally the basic ideas through some examples, then provide the formal definition; finally we state that the translation preserves static correctness.

The translation from Jam to Java must be defined in such a way to correspond to the informal Jam semantics we have illustrated in Sect.2. Hence, the two basic properties of mixins must be preserved, that is:

- the behavior of a class `H` obtained by instantiating a mixin `M` on a parent `P` must be "equivalent" to that of a class obtained by extending `P` by all the defined components of `M` (copy principle);
- mixin names can be used as reference types (independently from the existence of some mixin instance), and every class which is instance of a mixin must be subtype of both the mixin and the parent class.

The first point immediately gives an easy translation directive: every instantiation of a mixin `M` on a parent `P` must be expanded to a usual Java declaration of a class extending `P` and declaring all the defined components of `M` (plus the constructors possibly declared in the instantiation).

The second point is less trivial. Indeed, mixin types in Jam are a new kind of types, not existing in Java, hence they must be mapped onto either class or interface types.

A simple way to get the property that a mixin instance turns out to be a subtype of both the mixin and the parent type "for free" is to translate a mixin declaration by an interface declaration, and every instantiation by a Java class which (besides extending the parent) implements this interface; however, this choice introduces the problem that mixins in Jam can declare fields, while interfaces cannot.

On the other hand, translating a mixin declaration by a class declaration would have the advantage of making possible the declarations of fields, but would require to simulate in Java the implicit Jam type conversion from the mixin instance type to the mixin type.

Hence, we have adopted the first choice, solving the problem of field declarations by the standard technique of simulating fields by pairs of *accessor* methods for selecting (*getter*) and updating (*setter*) fields. For each field `f` in a mixin `M` declaration, the methods `M_$get$_f` and `M_$set$_f` are declared in the interface corresponding to the mixin declaration; then, in every class translating a mixin instance, `f` is declared as a field and the two methods are implemented in the obvious way[7].

[7] The implementation, for handling Java packages, uses a fully-qualified (as defined in 6.7 of [14]) mixin name, rewritten in some way to eliminate the dots, in the accessor name to avoid possible name clashes.

Let us now illustrate how the translation works in practice on the mixin Undo introduced in Sect. 1.

```
interface Parent$Undo {
 String getText() ;
 void setText(String s) ;
}
interface Undo extends Parent$Undo {
 String Undo_$get$_lastText() ;
 String Undo_$set$_lastText(String newValue) ;
 void setText(String s) ;
 void undo() ;
}
```

Fig. 12. Translation of a mixin declaration

Note that in the translation (shown in Fig. 12), together with the interface Undo corresponding to the mixin type, there is a second interface Parent$Undo which is extended by Undo and contains only the declarations of inherited methods.

This interface represents the translation of the type *Parent*(Undo) introduced in Sect. 2.4, that is, the generic parent type on which the mixin can be instantiated, and is necessary for the Java translation to correctly simulate the Jam extended rule for overloading resolution (an inherited method in a mixin M must be considered as it had been declared in a "generic" superclass of M, hence considered less specific of a defined method with the same signature).

We consider now an instantiation of the mixin Undo in Fig. 13, and the corresponding translation in Fig. 14.

```
class Example {
 String donald = "duck" ;
 String getText() { return donald ; }
 void setText(String donald) { this.donald = donald ; }
}
class ExampleWithUndo = Undo extends Example {}
```

Fig. 13. Undo instantiation example

As shown by the example, the class translating an instantiation of the mixin M on a parent P extends P and implements the interface M; moreover, the class contains a copy of all the fields and methods defined in M, including static members and abstract methods, and the implementation of the accessor methods for each field.

```
class Example { ... (unmodified) }

class ExampleWithUndo extends Example implements Undo {
 String lastText ;
 String Undo_$get$_lastText() { return lastText ; }
 String Undo_$set$_lastText(String newValue) {
  return lastText = newValue ;
 }
 void setText(String s) {
  lastText=getText() ; super.setText(s) ;
 }
 void undo() { setText(lastText) ; }
}
```

Fig. 14. Translation of a mixin instantiation

Note that, inside the mixin, no accessor invocation is used in the translation for accessing the fields. Such invocations are only necessary when accessing from external code. Moreover, using accessors is only needed when the field access is on an expression of the mixin type, while the code remains unchanged if the expression type is a mixin instance type. For instance, the following code would be kept as it stands by the translation process:

```
ExampleWithUndo e = new ExampleWithUndo() ;
System.out.println(e.lastText) ;
```

Inherited fields. Although inherited fields logically differ from defined fields, they are translated in exactly the same way: a pair of method accessors is generated.

Static fields. As shown in Sect.2.2, static fields do not belong to the mixin type. Therefore declarations of static fields within a mixin matter only for mixin instances. As a consequence, the pair of interfaces corresponding to a mixin does not contain any accessor for static fields. Instead, static fields will be inserted in every class corresponding to a mixin instance.

Formal translation. We formally define now the translation of Jam into Java. The aim is twofold: first, to define the dynamic semantics of Jam; second, to prove soundness of the Jam type system from the soundness of the Java type system [10] and from Theorem 1 which states that the translation preserves the static semantics.

As usual, for proving preservation of static correctness we need to provide a formal translation not only for Jam programs (environments and body declarations), but also for all judgments, hence for type assertions. In this paper, for lack of space, we omit translation clauses related to body declarations.

We denote by $[\![\, \Gamma \,]\!]$ the translation of a Jam environment Γ.

Since assertions in Γ may be mutually recursive, analogously to what happens for the static semantics, the translation of Γ (Fig.15) uses an auxiliary function taking an additional argument which is a larger environment.

$$\begin{array}{l}
[\![\Gamma]\!] = [\![\Gamma]\!]_\Gamma \\
[\![\gamma_1, \ldots, \gamma_n]\!]_\Gamma = \bigcup_{i \in \{1,\ldots,n\}} [\![\gamma_i]\!]_\Gamma
\end{array}$$

Fig. 15. Translation of environments

The translation of a Jam type assertion is a set of Java type assertions, and is defined in Fig.16. For all the type assertions γ for which there is no translation clause, we implicitly assume that $[\![\gamma]\!]_\Gamma = \{\gamma\}$.

$$\begin{array}{l}
[\![T \lhd_i^1 I]\!]_\Gamma = \{T <_i^1 I\} \text{ if } T \in \mathit{Mixins}(\Gamma) \quad [\![T \lhd_i I]\!]_\Gamma = \{T \leq_i I\} \text{ if } T \in \mathit{Mixins}(\Gamma) \\
[\![C \lhd_m M]\!]_\Gamma = \{C \lhd_i^1 M\} \\
[\![M \text{ is}_m\ \mathit{FST}\ \mathit{MST}\ \texttt{inherited}\ \mathit{FST}'\ \mathit{MST}']\!]_\Gamma = \\
\qquad \{M \text{ is}_i\ \mathit{Abstract}(\mathit{AccessorHeadings}(\mathit{FST}'[\mathit{FST}], M) \cup \mathit{MST}), \\
\qquad \mathit{Parent}(M) \text{ is}_i\ \mathit{Abstract}(\mathit{MST}'), M <_i^1 \mathit{Parent}(M)\} \\
[\![C \text{ is}_c\ K\ \mathit{KST}\ \emptyset\ \emptyset]\!]_\Gamma = \{C \text{ is}_c\ K\ \mathit{KST}\ \mathit{FSTMST}\} \\
\qquad \text{if } \Gamma \vdash C\ \lhd_m\ M, \Gamma \vdash M \text{ is}_m\ \mathit{FST}\ \mathit{MST}\ \texttt{inherited}\ \mathit{FST}'\ \mathit{MST}' \\
[\![C\ :\ \mathit{FST}_c\ \mathit{MST}_c]\!]_\Gamma = \{C : \mathit{FST}_c\ \mathit{MST}_c \cup \mathit{AccessorHeadings}(\mathit{FST}'[\mathit{FST}], M)\} \\
\qquad \text{if } \Gamma \vdash C\ \lhd_m\ M, \Gamma \vdash M \text{ is}_m\ \mathit{FST}\ \mathit{MST}\ \texttt{inherited}\ \mathit{FST}'\ \mathit{MST}' \\
[\![M\ :\ \mathit{FST}_m\ \mathit{MST}_m]\!]_\Gamma = \{M :\ \emptyset\ \mathit{Abstract}(\mathit{MST}_m \cup \mathit{AccessorHeadings}(\mathit{FST}'[\mathit{FST}], M))\} \\
\qquad \text{if } \Gamma \vdash M \text{ is}_m\ \mathit{FST}\ \mathit{MST}\ \texttt{inherited}\ \mathit{FST}'\ \mathit{MST}'
\end{array}$$

Fig. 16. Translation of type assertions

The translation of the type assertions having form $T \lhd_i^1 I$ and $T \lhd_i I$ depends on the type of the module T. If T is a mixin then the assertions are translated into subinterface assertions, otherwise (that is, if T is a class), they remain unaltered. The instantiation assertion becomes an implementation assertion. A mixin declaration is transformed into the declaration of two interfaces (the first being a subinterface of the second). A class declaration C is modified only in the case C is an instance of a mixin M; the translation in this case corresponds to the copy principle. Finally, type assignments for mixin instances and mixins are translated by introducing accessors. Of course, we assume that there are no name conflicts between accessors and user defined methods.

The function *Mixins* returns all the mixins declared in a given environment; the function *AccessorHeadings* returns the accessors (getter and setter) corresponding to a set of fields; the function *Abstract*, given a set of methods, returns as `abstract` methods all the instance methods.

We state now that the translation preserves the static semantics, in the sense that a statically correct Jam program is translated into a statically correct Java program. This is a implied by the more general property that every valid Jam judgment is translated into a set of valid Java judgments.

Theorem 1. *Let Γ be a well-formed Jam environment (that is, $\Gamma \vdash \diamond$ is valid), then:*

1. *for every valid judgment $\Gamma \vdash \gamma$, $[\![\gamma]\!]_\Gamma$ is well-defined and $[\![\Gamma]\!] \vdash [\![\gamma]\!]_\Gamma$ is valid.*

2. $[\![\Gamma]\!]$ *is a well-formed Java environment (that is,* $[\![\Gamma]\!] \vdash \diamond$ *is valid).*

The proof can be found in [1].

5 Implementation

The translator (called `jamc`), implemented in Java, performs a complete syntactic analysis and only a partial type-checking of Jam input source files. This means that every lexical or syntactic error in the source code will be detected by `jamc`, whereas most static errors will be found later by the Java compiler when trying to compile the Java source files produced by `jamc`. Hence obtaining bytecode from a Jam source involves the execution of two distinct tools: the translator and a standard Java compiler. The former requires the execution of a JVM, so this step might be quite expensive even though the translation process is cheap in computational terms. The cost of the latter step is, of course, the usual cost required by a standard Java program.

In the current implementation, we have chosen to follow the copy principle in a straightforward way, because this particular translation guarantees good performances. In other words, following the terminology introduced by Pizza [18], we have developed a *heterogeneous* translation which tends to favour the running time of the generated code penalizing the bytecode size. Alternatively, a *homogeneous* translation could be considered in order to share as much as possible code between the instantiations of a mixin, at the cost of some time overhead due to casts. These translations would be preferred in environments like embedded systems and smart-card applications where the code size is often a main concern.

We finally mention that `jamc` must deal with two features that we have not discussed in this paper: packages and access modifiers. For more details see [1].

6 Related and Further Work

In the preceding sections we have described Jam, a smooth extension of Java supporting mixins, and we have formally defined its static semantics and a translation into Java. The latter has been implemented by a Jam to Java translator which makes Jam executable on every platform implementing a Java Virtual Machine. In this last section, we provide some detailed comparison with related work and discuss some alternative design choices and directions for further investigations.

Object oriented languages supporting mixins. To our knowledge, the only existing proposals for extensions of object-oriented languages with mixins are [7] and [13].

In [7] is presented an extension of Smalltalk with mixins. The design principles of this extension are very similar to those we have followed in Jam. Indeed, mixins are seen as functions from superclasses into heir classes, instantiation is possible only if the candidate parent class contains all the methods invoked via `super` in the mixin, mixins do not influence the behavior of existing Smalltalk programs, hence the extension is fully upward-compatible. The most significant difference is due to the fact that Smalltalk is untyped, and so, most of the problems we had to face in the design of Jam simply do not exist for Smalltalk; the most remarkable of these problems is that mixins introduce a new kind of reference type. As in our approach (see Sect.2.3) overriding takes place uniformly both for methods which are invoked via `super` and for others. Following our same principle that mixin instantiation should produce a correct heir class, the candidate parent class must not contain instance variables with the same name of some defined in the mixin (indeed in Smalltalk hiding parent variables is forbidden). Moreover, mixins can be easily eliminated from a program by automatically creating a class for each mixin invocation and duplicating the mixins code for it (in other words, mixins have a pure copy semantics, corresponding to β-rule for function application), while for Jam this is not enough since mixins are *types* so they cannot be just eliminated.

In [7], a mixin can be composed with another mixin (the expected semantics is exactly function composition) and a mixin can also be "extracted" from an existing class: in this case, its components are those declared in the class. Both the possibilities seem very useful and adding them to Jam will be matter of further work, even though a generalization allowing full mixin composition seems in the Java case not trivial, on both design and implementation side.

The authors have developed a working extension which has been used for real applications.

In [13] is described MixedJava, a theoretical language which has a Java-like syntax where it is only possible to declare either mixins or interfaces, while usual classes are seen as particular mixins which define all the components.

In MixedJava, there are two kinds of mixins.

- *Atomic* mixins, whose declaration, similar to that of a usual Java class, contains fields, methods and an interface which specifies the expected superclass. This interface plays the same role as the `inherited` part of mixins in Jam, with the difference that it must be explicitly declared by the programmer, while in Jam the interface is created during the translation process.
  A basic difference (see Sect.2.3) is that in mixin instantiation (which in MixedJava is just a special case of mixin composition, see below) methods in the heir override methods in the parent *only if* they are explicitly mentioned in the inheritance interface, while in case of unexpected overriding both the versions are kept.
- *Compound* mixins, roughly based on function composition, as happens for the Smalltalk extension described above, but actually more involved, for the constraints on method overriding explained above.

The work presented in [13] differs significantly from ours. Namely:

- The proposed language is theoretical, while Jam is designed to be a working upward-compatible extension of Java (1.0).
- In MixedJava inherited components can be only methods, since they are specified via an interface. The authors motivate this choice by the consideration that programming via interfaces is cleaner; in Jam, we have chosen

as the main principle that mixins should be similar to usual heir classes as much as possible.

- In Jam mixins can be only instantiated on classes, and there is no notion of mixin composition. As already stated, this is an important possibility of extension of Jam to be investigated in the future.
- As mentioned above, MixedJava adopts an ad-hoc solution to unexpected overriding, while in Jam methods in the parent class are uniformly overridden by methods in the heir class. This different policy is probably the most important difference between the two approaches. A disadvantage of our approach is that if the parent class incidentally has some method which is in conflict with one defined in the mixin, it is left to the user to avoid this instantiation (hence the mixin becomes useless for this particular case) or to get an heir class with some overriding which was not planned when designing the mixin. However, the conflicts resolution in [13], essentially based on the idea of keeping both method versions, leads to ambiguity problems which are typical of multiple inheritance (a class inherits two different definitions for the same method), heavily complicating both language semantics and a possible implementation (only outlined in [13]). A future development could be the analysis of intermediate solutions.

Mixins vs. parametric types. A great effort has been spent in the last years by the scientific community in proposing extensions of Java with parametric types, see, e.g., Pizza [18], its evolution GJ [9] and PolyJ [16]; these proposals are presently under consideration of the Java Community Process initiative. With respect to Java extensions with parametric types, extensions with mixins go in a different, in a sense orthogonal, direction. The main disadvantage of the mixin-based approach is that there is no mean in a mixin to refer either to the generic parent class to which the mixin will be applied or to the generic heir class obtained by instantiation, as illustrated in Sect. 2.6.

Hence, there are cases where heir classes cannot be "abstracted" in a mixin definition. Introducing canonical notation for the parametric names of the parent and heir class, say `P*` and `H*`, respectively, we could transform the class `H` shown in Sect. 2.6 in a mixin `M` as follows.

```
mixin M {
 inherited static int counter ; static int counter ;
 static void incrThat() { ++P*.counter ; }
 ...
 int value ;
 public boolean equals(Object that) {
  if (that instanceof H*) return ((H*)that).value == value ;
  return false ;
 }
}
```

Obviously, in this case the copy principle should be modified, saying that a class `H = M extends P` should be equivalent to a class extending `P` and containing the definitions in `M` where all the occurrences of the parametric names `P*` and `H*` have been replaced by `P` and `H`, respectively. Introducing this possibility would allow a form of parametric polymorphism (limited to heir classes), similar to the extensions of Java with parametric types mentioned above. However, with this choice we would lose one of the two design principles of Jam, that is, the fact that a mixin name can be used as a type. Indeed, in the approaches based on parametric types, these cannot be directly used as Java types, but are type schemata; hence it is not possible to uniformly use all their instantiations. It is not clear whether (and how) it is possible to reconcile these two different ways of achieving abstraction: parametric modules (class-to-class functions) where this parametricity is fully exploited, and using modules as types. The problem is not trivial and deserves further investigation.

Flexible matching. Assume that P is a supertype of H and consider the following declarations.

```
mixin M {                            class C1 {
 inherited void f(H, H) ;             void f(P p, H h) {}
}                                    }
```

In Jam instantiating M on C1 is illegal, since C1 does not provide an implementation for the method void f(H,H). Indeed, the matching between the inherited methods and the corresponding methods in the parent class is required to be *exact* (same arguments and return type, and equivalent throws clause). An interesting possibility, which could be matter of a future extension, could be a *flexible* matching, allowing contravariance on arguments type and covariance on return type. However, note that the exception types must be invariant (modulo the equivalence $=_e$) in order to preserve the soundness of the type-system.

Allowing this flexibility, C1 turns out to be a correct parent class for M. However, this kind of matching leads to some new problems w.r.t. the exact matching case. Let us consider this other class declaration.

```
class C2 {
 void f(P p, H h) {}
 void f(H h, P p) {}
}
```

In this case, assuming that we want to instantiate M on C2, we have to decide which of the two methods declared in C2 to use as implementation of the inherited method in M. The choice could either be driven, in analogy with the overloading resolution in Java, by the notion of most specific applicable method[8], or left to the user via a mechanism which permits to explicitly specify in the instantiation the association of inherited methods with those defined in the parent class. Further enhancements could also include a mean of renaming methods to make them match an inherited specification.

Shared static components. In Sect. 2.2, we have seen that each class has its own copy of the static components declared in the mixin. As already mentioned

[8] Hence, in this particular example, there would be an error.

there, other two design choices are conceivable: either mixin instances share a unique copy for each static component (in this way they would be part of the mixin type), or leave to the user, by means of a keyword `shared` or analogous mechanism, the choice between the two options. The latter choice, which has some appeal, would require the introduction of some constraint, for instance the fact that a `shared static` method may not invoke a `static` method.

Acknowledgments. We warmly thank Sophia Drossopoulou and the anonymous referees for the careful reading and useful suggestions for improving the paper.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam - A smooth extension of Java with mixins. Technical report, DISI-TR-99-15, University of Genova, November 1999. Available at `http://www.disi.unige.it/ftp/person/AnconaD/Jam.ps.gz`.
2. D. Ancona and E. Zucca. A theory of mixin modules: basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
3. D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *Principles and Practice of Declarative Programming, 1999*, Lecture Notes in Computer Science, pages 62–79. Springer Verlag, 1999.
4. G. Banavar and G. Lindstrom. An application framework for module composition tools. In *ECOOP '96*, number 1098 in Lecture Notes in Computer Science, pages 91–113. Springer Verlag, July 1996.
5. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
6. G. Bracha and W. Cook. Mixin-based inheritance. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1990*, pages 303–311. ACM Press, October 1990. SIGPLAN Notices, volume 25, number 10.
7. G. Bracha and D. Griswold. Extending Smalltalk with mixins. In *OOPSLA96 Workshop on Extending the Smalltalk Language*, April 1996. Electronic note available at `http://www.javasoft.com/people/gbracha/mwp.html`.
8. G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, April 1992. IEEE Computer Society.
9. G. Bracha, M. Odersky, D. Stoutmire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, October 1998. Home page: `http://www.cs.bell-labs.com/who/wadler/pizza/gj/`.
10. S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, number 1523 in Lecture Notes in Computer Science, pages 41–82. Springer Verlag, Berlin, 1999.
11. S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Dept. of Computing - Imperial College of Science, Technology and Medicine, October 1999.
12. D. Duggan and C. Sourelis. Mixin modules. In *Intl. Conf. on Functional Programming*, pages 262–273, Philadelphia, June 1996. ACM Press. SIGPLAN Notices, volume 31, number 6.

13. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM Symp. on Principles of Programming Languages 1998*, pages 171–183, January 1998.
14. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.
15. S.C. Keene. *Object Oriented Programming in Common Lisp: A Programming Guide in CLOS.* Addison-Wesley, 1989.
16. A.C. Meyers, J.A. Bank, and B. Liskov. Parameterized types for Java. In *Proc. 24th ACM Symp. on Principles of Programming Languages.* ACM Press, January 1997.
17. D.A. Moon. Object oriented programming with Flavors. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1986*, pages 1–8, 1986.
18. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *ACM Symp. on Principles of Programming Languages 1997.* January 1997.
19. A. Snyder. CommonObjects: An overview. *SIGPLAN Notices*, 21(10):19–28, 1986.

A Mixin-Based, Semantics-Based Approach to Reusing Domain-Specific Programming Languages

Dominic Duggan

Department of Computer Science
Stevens Institute of Technology
Hoboken, New Jersey 07030.
dduggan@cs.stevens-tech.edu

Abstract. Domain-specific programming languages (DSLs) are emerging as an important paradigm for the engineering of large reliable software systems. Modular interpreters are an approach to building off-the-shelf software components that implement fragments of DSLs. We describe an approach to implementing modular interpreters in an object-oriented fashion, using the design pattern of *extensible computations*. The modular structuring and reuse of DSL implementations has potentially important lessons for object-oriented reuse, because of the highly recursive nature of any non-trivial programming language, and the close semantic relationship between inheritance and recursion. We give paradigmatic examples of the definition of extensible computations in the Java programming language, and consider what extensions would be required for typed object-oriented languages in order to support this approach in a statically type-safe manner.

Keywords: Domain-specific programming languages, modular interpreters, monads, mixin-based inheritance, Java.

1 Introduction

Domain-specific programming languages (DSLs) have recently gained attention as being particularly important in the software engineering process [10,38,40]. A problem with developing DSLs is their high start-up cost, due to the difficulty with implementing a new DSL implementation every time one is needed. One possible solution to this problem is to develop an application framework for developing DSLs. Such an application framework would provide modular implementations of reusable domain-specific programming languages. An interpreter for a DSL would be provided as a modular building block. The application developer who desired a DSL with particular features could then combine the corresponding building blocks into a DSL with the required semantics.

This paper describes such an approach to developing reusable implementations of domain-specific programming languages. The approach is based on developing modular interpreters for programming language fragments, as an object-oriented application framework, and then allowing these modular interpreters to be combined to provide an off-the-shelf implementation of a DSL providing particular features. The modular structuring and reuse of DSL implementations has potentially important lessons for object-oriented reuse, because of the highly recursive nature of any non-trivial programming language, and the close semantic relationship between inheritance and recursion.

The technical challenge in achieving this objective, is to be able to define modular interpreters for programming language fragments in such a manner that no global assumptions need to be made about the form of any final DSL in which such fragments are

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 179–200, 2000.
© Springer-Verlag Berlin Heidelberg 2000

combined. For example an interpreter for a language fragment comprising exceptions (which might be written in a continuation-passing style) should be defined completely independently of an interpreter for a language fragment with functions (requiring environments in the interpreter).

We describe an approach to doing this, using a new design pattern called *extensible computations*. Extensible computations are based on the idea of monads from categorical semantics. The use of monads to structure interpreters in a modular fashion has already been investigated. The use of "monad transformers" (mapping from monads to monads) to describe composable modular interpreters has been described [30]. In contrast to the aforesaid work, extensible computations are based on a particular form of inheritance, defining modular interpreters as *mixin classes* and relying on a particular form of *mixin-based inheritance* to compose these interpreters. We provide a more complete comparison with monad transformers in Sect. 7.

In Sect. 2 we introduce our approach to defining modular interpreters. We give paradigmatic examples of the definition of modular interpreters in Sect. 3, Sect. 4 and Sect. 5. In Sect. 6 we describe the metalanguage support suggested by modular interpreters, and briefly describe an extension of the Java language that provides this support. We consider related work in Sect. 7, while Sect. 8 provides our conclusions.

2 Modular Interpreters

The key to our approach is to define a common interface for the evaluator function in every interpreter fragment. The essential idea is to regard the interpreter as mapping from an expression to a computation, so every interpreter building block defines an instance of an evaluation function of type:

```
public class Interp {
   public interface Expr {
      public Computation eval ();
   };
```

A computation in turn is defined as a mapping from an *input context* to an *output context* and a value:

```
   public interface Computation {
      public ValueOutputContext run (InputContext ic);
   };
```

The input and output contexts are the inputs to, and outputs from, respectively, the execution of a computation. In this abstract class they are empty; we expect concrete classes that extend this to extend the input and output contexts as appropriate. Therefore the input to a computation has type `InputContext`, a type that refers to the final input context type (after extensions). Similarly the output from a computation has type `OutputContext`.

```
  public interface Value { };
  static public class InputContext { public InputContext next; };
  public InputContext inputContextFactory () {
    return nextInterp.inputContextFactory ();
  };
  static public class OutputContext { public OutputContext next; };
  public OutputContext outputContextFactory () {
    return nextInterp.outputContextFactory ();
  };
```

```
static public final class ValueOutputContext {
  public Value v;
  public OutputContext oc;
  public ValueOutputContext (Value v, OutputContext oc) {
    this.v = v; this.oc = oc;
  }
}
```

For example, the input context for the interpreter for functions contains an environment giving the bindings of the non-local variables, while the output context is empty. For an interpreter for a language with imperative constructs, the input context would be an input store (in the standard way of defining the semantics of an imperative language), while the output context would be the store updated by the computation. The challenge in combining interpreter fragments is to ensure that the input and output contexts, and the ways of defining them for each construct, are suitably combined.

Interpreter mixins are defined in such a way that they facilitate the composition of the computations defined in each module. Here we adapt a notion from denotational semantics, that of *monads*. A monad is essentially a categorical abstraction of a computation. The particular form of monads used by computer scientists are Kleisli triples, and are composed of a type (the type of a computation), a "unit" operation for injecting a value into a computation, and a "bind" operation for binding a variable to a value in a computation, and composing this with a computation that produces that value. The use of monads to modularize denotational semantics was first advocated by Moggi [36]. Wadler [42] popularized monads as a way of writing interpreters in a modular style, although he did not consider how to combine monadic definitions of modular interpreters. The relevance of monads is that they explicate the injection and composition operations of a computation. The injection operation injects a value into a computation, and forms the basis for defining a new computation.

```
public Computation unit (Value v) {
  return nextInterp.unit (v);
};
```

The composition operation joins two already-defined computations into a new computation; essentially this forms the sequential composition of two computations, where the result of the first computation is fed as the input to the second computation.

```
public interface Continuation {
  public Computation kontinue (Value v);
};
public Computation compose (Computation left, Continuation right) {
  return nextInterp.compose (left, right);
};
```

A default definition of these methods is provided in Fig. 1. These defaults are sufficient provided the interpreter does not need to handle exceptional control constructs; if the latter is the case, the definitions of `unit` and `compose` must be redefined to give a continuation-passing semantics.

`unit` and `compose` must be modified in each interpreter mixin to properly thread input and output contexts through evaluation. To facilitate these modifications, `unit` and `compose` are defined in terms of the following methods. These methods comprise the "hooks" for each interpreter module to modify the semantic glue as appropriate:

```
  public OutputContext outputContext
         (Value v, InputContext ic) {
         return nextInterp.outputContext (v, ic);
  };
  public InputContext leftInputContext
         (Continuation right, InputContext ic) {
         return nextInterp.leftInputContext (right, ic);
  };
  public InputContext rightInputContext
         (InputContext ic, OutputContext leftOutputCtxt) {
         return nextInterp.rightInputContext (ic, leftOutputCtxt);
  };
```

Finally we build an interpreter by composing a sequence of interpreter building blocks, each one delegating to the next building block for functionality it does not provide. To describe this delegation we simulate a notion of "super" (different from Java super because of the way it is bound) using a variable `nextInterp` that points to the next building block:

```
  protected Interp nextInterp;
  public Interp () { nextInterp = null; }
  public Interp (Interp n) { nextInterp = n; }
```

The `root` variable is used to record the root of a composition of interpreters. All calls to interpreter functions should be through this root pointer, since it allows access to the final result of composing the definitions of these functions in each mixin.

```
  protected Interp root;
  public void setRoot () { setRoot(this); }
  public void setRoot (Interp n) {
    root = n;
    if (nextInterp != null) nextInterp.setRoot (n);
  }
};
```

Default definitions for these semantic glue functions are provided in Fig. 1. Using these definitions, and assuming interpreter mixins for functions (Function), imperative constructs (Imperative) and exceptional control (Control), an interpreter for a language with functions, assignment and exceptions can be composed using the following step:

```
Defaults interpDef = new Defaults ();
Imperative interpImp = new Imperative (interpDef);
Function interpFunc = new Function (interpImp);
Control interpCtl = new Control (interpFunc);
Interp interp = interpCtl;
interp.setRoot ();
```

Then an interpreter can be executed by building an AST and executing the `eval` method of the AST, and then running the resulting computation:

```
Interp.Expr func = interpFunc.makeAbs ("x", interpFunc.makeVar ("x"));
Interp.Expr app = interpFunc.makeApp (func, func);
Interp.InputContext ic = interp.inputContextFactory ();
Interp.ValueOutputContext voc = app.eval().run(ic);
Interp.Value value = voc.v;
```

In the next three sections we give examples of interpreter building blocks, defined in a domain-specific metalanguage intended for specifying such building blocks. Each interpreter building block must specify several things:

```
public class Defaults extends Interp {

  public Computation unit (final Value v) {
    return new Computation () {
      public ValueOutputContext run (InputContext ic) {
        return new ValueOutputContext (v, root.outputContext (v, ic));
      } };
  };

  public Computation compose (final Computation left,
                              final Continuation right) {
    return new Computation () {
      public ValueOutputContext run (InputContext ic) {
        ValueOutputContext voc =
             left.run (root.leftInputContext (right, ic));
        return right.kontinue(voc.v).run
                             (root.rightInputContext (ic, voc.oc));
      } };
  };

  public OutputContext outputContext
         (Value v, InputContext ic) {
    OutputContext oc = root.outputContextFactory ();
    return oc;
  };
  public InputContext leftInputContext
         (Continuation right, InputContext ic) {
    InputContext ic2 = root.inputContextFactory ();
    return ic2;
  };
  public InputContext rightInputContext
         (InputContext ic, OutputContext leftOutputCtxt) {
    InputContext ic2 = root.inputContextFactory ();
    return ic2;
  };

  public InputContext inputContextFactory () {
    return new InputContext ();
  };
  public OutputContext outputContextFactory () {
    return new OutputContext ();
  };

};
```

Fig. 1. Default Definitions for Interpreter Mixin Glue

1. the abstract syntax of programs in the object language;
2. the values computed by the interpreter;
3. the types of the input and output contexts involved in a computation;
4. the evaluation rules for the interpreter itself (one evaluation rule for each object language construct); and
5. the "glue" for composing computations, and for injecting values into computations.

3 A Language with Functions

As a first example, here is the definition of an interpreter building block for functions. The syntax of this language might be described by:

$$e ::= x \quad | \quad \mathrm{func}(x)e \quad | \quad e_1(e_2)$$

The description of the interpreter is given by:

```
public class Function extends Interp {

  public class Env {
    public Value lookup (String var) { ... };
    public void pushBinding (String var, Value binding) { ... };
    public void popBinding () { ... };
  };

  public class InputContext extends Interp.InputContext {
     public Env env;
     public InputContext () { this.env = new Env(); }
  };
```

This interpreter contains a member class for environments, and the input context includes a field for an environment that is part of the input to evaluation. inputContextFactory is defined for this module to create an instance of Function.inputContext, with further fields in the input context provided by the next field. The setEnv and getEnv fields are applied through an arbitrary input context, and use dynamic type tags and downcasting to search for the input context component containing the environment field. The output context is empty, so this interpreter module relies on the default outputContextFactory definition (inherited from Interp) simply delegates to the next interpreter module:

```
   public Interp.InputContext inputContextFactory () {
      Interp.InputContext ic = new InputContext ();
      ic.next = nextInterp.inputContextFactory ();
      return ic;
   };
   public void setEnv (Interp.InputContext ic, Env env) {
      if (ic instanceof InputContext) {
         ((InputContext)ic).env = env;
      } else {
         setEnv (ic.next, env);
      }
   };
```

```
public Env getEnv (Interp.InputContext ic) {
   if (ic instanceof InputContext) {
      return ((InputContext)ic).env;
   } else {
      return getEnv (ic.next);
   }
};
```

The outputContext operation used in the definition of unit, and the leftInputContext and rightInputContext operations used in the definition of compose, are at the core of our approach to extending computations. They represent the hooks in the definitions of the interpreter building blocks that allow contexts to be correctly threaded through the execution of the interpreter. They are fragments in the definition of context-building operations, whose final definitions are obtained by combining the fragments specified in each interpreter module. For example the definition of outputContext in this module is empty. However in the next section we give another module which has a non-empty outputContext definition. If these two interpreters are combined, the definition of outputContext in the result should contain the fragment of outputContext specified in the latter module. In this interpreter module, leftInputContext and rightInputContext are defined to set the environment field of the corresponding computation's input context to the environment field of the overall input context:

```
public Interp.InputContext leftInputContext
      (Continuation right, Interp.InputContext ic) {
   Interp.InputContext left = nextInterp.leftInputContext(right,ic);
   setEnv (left, getEnv (ic));
   return left;
};
public Interp.InputContext rightInputContext
      (Interp.InputContext ic, Interp.OutputContext leftOutputCtxt) {
   Interp.InputContext right =
      nextInterp.rightInputContext(ic, leftOutputCtxt);
   setEnv (right, getEnv (ic));
   return right;
};

public interface Closure extends Value {
   public Computation exec (Expr arg);
};
```

There is a single value type, the type of closures (which map from an argument expression to a computation).

Definitions of input and output contexts contributed by each mixin are combined using the next instance variable defined in the Interp.InputContext and Interp.OutputContext base classes, while the definitions of the above operations contributed by each mixin are composed using the nextInterp instance variable defined in the Interp base class. The semantic operations that invoke these operations refer to the result of combining these fragments by invoking these operations from root, the root of the list of interpreter building blocks.

To give the definition of the evaluator, for variables we simply look up the variable in the environment, and inject the resulting value into the computation:

```
public class Var implements Expr {
  public final String x;
  public Computation eval () {
    return new Computation () {
      public ValueOutputContext run (Interp.InputContext ic) {
        Computation comp = root.unit (getEnv(ic).lookup(x));
        return comp.run (ic);
      }
    };
  }
};
```

The value is injected into the computation using the unit operation. This builds a computation that expects an input context. Looking up the variable also requires accessing the input context, and the result of this lookup is the argument to unit. Therefore we build a computation that accesses the input context to perform the lookup, then uses unit to build a computation from the resulting value, then immediately runs the resulting computation on the current input context.

For applications, we evaluate the rator, and compose the resulting computation with a computation that evaluates the closure:

```
public class App implements Expr {
  public final Expr rator;
  public final Expr rand;
  class K implements Continuation {
    public Computation kontinue (Value v) {
      Closure cl = (Closure) v;
      return cl.exec (rand);
    }
  };
  public Computation eval () {
    return root.compose (rator.eval (), new K ());
  }
};
```

For abstractions, we build a closure that evaluates the rand and then evaluates the function body in the original environment (ignoring the call-site environment, but potentially using any other elements of the call-site context):

```
public class Abs implements Expr {
  public String formal;
  public Expr body;
  public Computation eval () {
    return new Computation () {
      public ValueOutputContext run (Interp.InputContext ic) {
        Computation comp =
            root.unit (new CBVClosure (Abs.this, getEnv(ic)));
        return comp.run (ic);
      }
    };
  }
};
```

The implementation for closures is provided by the `CBVClosure` class, that evaluates a function argument at the point where the function is called:

```
  public class CBVClosure implements Closure {
    public Abs function;
    public Env defSiteEnv;
    public CBVClosure (Abs e, Env env) { function = e; defSiteEnv = env; }
    class K implements Continuation {
      public Computation kontinue (final Value v) {
        return new Computation () {
          public ValueOutputContext run (Interp.InputContext ic) {
            Env saved = getEnv(ic);
            defSiteEnv.pushBinding (function.formal, v);
            setEnv(ic, defSiteEnv);
            ValueOutputContext result = function.body.eval().run(ic);
            setEnv(ic, saved);
            defSiteEnv.popBinding ();
            return result;
          }
        };
      }
    }
    public Computation exec (Expr arg) {
      return root.compose (arg.eval (), new K ());
    };
  }
```

Here `defSiteEnv` is the context where the closure is created, while `getEnv(ic)` is the context where the closure is evaluated (i.e. the call-site context). At the point where the evaluator is recursively called to evaluate the closure body, the environment passed to that evaluation is the definition-site environment with the formal parameter bound to the value of the actual. The update binds the environment component in the input context to closure evaluation (`ic`), to the definition-site environment (`defSiteEnv`) extended with the binding of the formal parameter. This is the usual static-scope semantics for procedure evaluation.

4 An Imperative Language

As our second example, we consider an interpreter building block for imperative constructs. The mini-language might have the abstract syntax:

$$e ::= \mathrm{ref}(e) \quad | \quad !e \quad | \quad e_1 := e_2 \quad | \quad (e_1; e_2)$$

The interpreter module for this mini-language defines the clauses in the abstract syntax and value types:

```
public class Imperative extends Interp {
  public class Location implements Value { int location; }
  public class Store {
    public Store update (int location, Value v) { ... };
    public Value access (int location) { ... };
  };
```

```
public static class InputContext extends Interp.InputContext {
  public Store store;
  public InputContext() { store = new Store(); }
};
public Interp.InputContext inputContextFactory () {
  Interp.InputContext ic = new InputContext ();
  ic.next = nextInterp.inputContextFactory ();
  return ic;
};
public static class OutputContext extends Interp.OutputContext {
  public Store store;
  public OutputContext() { store = new Store(); }
};
public Interp.OutputContext outputContextFactory () {
  Interp.OutputContext oc = new OutputContext ();
  oc.next = nextInterp.outputContextFactory ();
  return oc;
};
```

This defines a language of mutable reference cells (as is found in ML [34]). A reference cell is created with the `ref` constructor, and explicitly dereferenced with `!`. The semantics of imperative languages involves threading a "store," an abstraction of computer memory, through the execution of the interpreter. In this semantics, the `ref` operation allocates a new location in memory, while assignment updates the contents of a location in memory.

We omit the details of the semantics for the imperative constructs (see [16] for the details in a functional metalanguage). We need to define the cases for threading contexts through an interpreter that includes this building block:

```
public Interp.OutputContext outputContext
       (Value v, Interp.InputContext ic) {
   Interp.OutputContext oc = nextInterp.outputContext (v, ic);
   setStore (oc, getStore (ic));
   return oc;
};
public Interp.InputContext leftInputContext
       (Continuation right, Interp.InputContext ic) {
   Interp.InputContext left = nextInterp.leftInputContext(left,ic);
   setStore (left, getStore (ic));
   return left;
};
public Interp.InputContext rightInputContext
       (Interp.InputContext ic,
        Interp.OutputContext leftOutputCtxt) {
   Interpreter.InputContext right =
      nextInterp.rightInputContext(ic,leftOutputCtxt);
   setStore (right, getStore (leftOutputCtxt));
   return right;
};
```

Injecting a value into a computation simply threads the store in the input context into the output context. For composition, the store in the input context of the left computation is taken from the store in the overall input context, while the store in the input context of the right computation is taken from the output context of the left computation. The default definitions of unit and compose are sufficient for this interpreter.

5 A Language with Exceptional Control

As a final but important example, consider a language with control constructs such as gotos, coroutines, exceptions, etc. The semantics for all of these constructs requires that the semantics be defined in continuation-passing style, where a continuation is intuitively a formalization of the program control stack. A non-local transfer of control can then be effected by "throwing" the continuation (intuitively over-writing the current control stack with the saved control stack). So we have:

$$e ::= \texttt{callcc}(\lambda x.e) \quad | \quad \texttt{throw}(e, e')$$

The `callcc` ("call with current continuation") construct stores the current continuation into a program variable x, so this interpreter fragment introduces a variable-binding construct. In the semantics, the input context must include an environment recording the run-time bindings of variables. So this interpreter must share an implementation of environments with the interpreter for functions. This can be done by factoring the implementation of environments into a separate module.

```
public class InterpEnv {
  public static class InputContext extends Interp.InputContext {
    public Env env;
    public InputContext () { this.env = new Env(); }
  };
  public static class Env { ... }
  public static void setEnv (Interp.InputContext ic, Env env) { ... }
  public static Env getEnv (Interp.InputContext ic) { ... }
}
```

The `Function` mixin must be restructured to factor out the implementation of environents:

```
public class Function extends Interp {

  public Interp.InputContext inputContextFactory () {
    Interp.InputContext ic = new InterpEnv.InputContext ();
    ic.next = nextInterp.inputContextFactory ();
    return ic;
  };

  public Interp.InputContext leftInputContext
      (Continuation right, Interp.InputContext ic) {
    ... InterpEnv.setEnv (left, InterpEnv.getEnv (ic)); ...
  };
  public Interp.InputContext rightInputContext
        (Interp.InputContext ic, Interp.OutputContext leftOutputCtxt) {
     ... InterpEnv.setEnv (right, InterpEnv.getEnv (ic)); ...
  };
```

The `Control` mixin similarly relies on the factored definition of environment-containing input contexts. This mixin defines a value type for continuation values, that represent the result of reifying continuations into object language programs. This mixin also defines input contexts as an extension of the input context defined by InterpEnv:

```
public class Control extends Interp {
  public interface Cont extends Value {
    public ValueOutputContext returnk (ValueOutputContext result);
  };

  public static class InputContext extends InterpEnv.InputContext {
    public Cont cont;
  };
```

Unlike Function and Imperative, which rely on the default definitions of unit and compose (corresponding to “direct” semantics), the Control module must redefine the definitions of these operations to use a “continuation-passing” style. This makes the order of composition of interpreter fragments significant: the fragment for continuations must be combined to the left of the fragment that provides the original definition of these operations, and overrides the previous definitions. Therefore any references to root.unit and root.compose in any interpreter module will pick up the definitions provided by the Control module:

```
  public Computation unit (final Value v) {
    return new Computation () {
      public ValueOutputContext run (Interp.InputContext ic) {
        ValueOutputContext voc =
          new ValueOutputContext (v, root.outputContext (v, ic));
        return getCont(ic).returnk (voc);
      }
    }
  };

  public Computation compose
       (final Computation cl, final Continuation cr) {
    return new Computation () {
      public ValueOutputContext run (Interp.InputContext ic) {
        return cl.run(root.leftInputContext (cr, ic));
      }
    }
  };
```

The outputContext operation does nothing for this interpreter module, while the leftInputContext and rightInputContext operations build up the continuation in the input context. Evaluation of callcc receives the call-site continuation from the input context, builds a value from it, injects this value into a computation, and then composes this computation with the evaluation of the argument to callcc, effectively “feeding” the call-site continuation as a value to the program. Throwing evaluates the rator continuation, evaluates the value to be passed to the continuation, and then invokes the continuation tail-recursively.

6 Requirements of a Metalanguage

We have provided the definitions of the interpreter building blocks using Java, resorting to programming conventions and run-time casting to overcome the absence of support for mixin-based inheritance in Java. In this section we consider the language support that this application reveals would be useful in any metalanguage for implementing modular interpreters[1].

The metalanguage must support a particular form of mixin-based inheritance. Although aspects of this metalanguage have been investigated in the literature, there has been little investigation of the combination of these approaches in typed languages. Bracha and Lindstrom considered a language of "modules," where a module is a collection of mutually recursive mixin classes [7], however their language was an untyped language of records and functions. Duggan and Sourelis introduced *mixin modules* [17,18], which combine both mutually recursive types and mutually recursive functions, and allow these type and function definitions to be combined with definitions in other modules. The motivation for this work was shortcomings with module interconnection languages for programming-in-the-large, where linking was based only on instantiating parameterized modules [5]. Ernest [19] introduces gbeta, an extension of Beta that provides the combination of mixins and virtual patterns, as well as a static analysis to ensure that the combinations of these mechanisms is well-formed.

None of these works considers how to incorporate this extension into Java. Just as recent work on mutually recursive self types has been motivated by the desire to provide a functionality analogous to Beta patterns in Java [8], this section considers how an analogous combination of mixins and mutually recursive self types could be incorporated into Java. The presentation is informal, relying on the encodings of modular interpreters in earlier sections to exemplify the semantics of mixin combination considered here.

There are two extensions that we require to Java. First, we require a generalization of inheritance, that allows a collection of mutually recursive classes to be simultaneously extended through inheritance. This extension must allow references in classes to other classes, to be rebound to the extensions of those other classes after inheritance. We use a Java extension proposed by Bruce et al [8]. Here is a simple example:

```
public class SubjectObserver {
  public static class Observer as ThisObserver {
    public void notify (@ThisSubject s) { ... };
  };
  public static class Subject as ThisSubject {
    protected @ThisObserver observers[];
    public void notifyObservers () {
      ... observers[i].notify(this) ...
    };
  };
}
```

This provides an implementation of the Subject-Observer pattern, as a class that provides two mutually recursive nested classes, Observer and Subject. These two classes do not refer to each other directly, but rather the references are indirect through the *self types* ThisObserver and ThisSubject, respectively. The latter are type variables

[1] It should be noted that many design patterns also require constructs that are not provided in current object-oriented languages, although recent research is addressing this issue [8].

that vary with the type of “self,” and so may have different instantiations depending on the final classes in which the objects are created. For example, if we define:

```
public class WindowSubjectObserver extends SubjectObserver {
  public static class WindowObserver extends Observer { ... };
  public static class WindowSubject extends Subject { ... };
}
```

Then in this subclass, the `notify` method for observers is defined for `WindowSubject` objects, while a subject has a list of `WindowObserver` objects. The `@` notation is a technical device, requiring “exact” types for the notify parameter and the list of observers. See Bruce et al [8] for further details.

The other Java extension we require is mixin-based inheritance. While mixin-based inheritance for typed object-oriented languages is still an avenue of research [6,7,23,3], we will assume the following extensions to Java:

$$
\begin{aligned}
\mathit{decl} &::= [\ \texttt{abstract}\ ]\ \texttt{mixin}\ \mathit{identifier}\ \mathit{mixdecl}\ \{\ \mathit{decl}_1;\ \ldots\ \mathit{decl}_n;\ \} \\
\mathit{mixdecl} &::= [\ \texttt{implements}\ \mathit{identifier}_1, \ldots, \mathit{identifier}_k\ ]\ [\ \texttt{extends}\ \mathit{identifier}\ ] \\
&\qquad\qquad [\ \texttt{expects}\ \mathit{identifier}\ ] \\
\mathit{decl} &::= \texttt{mixin}\ \mathit{identifier}\ \texttt{combines}\ \mathit{identifier}_1,\ \ldots,\ \mathit{identifier}_n \\
\mathit{expr} &::= \texttt{inner}(\mathit{expr}_1, \ldots, \mathit{expr}_k) \\
\mathit{expr} &::= \texttt{inner}.\mathit{identifier}(\mathit{expr}_1, \ldots, \mathit{expr}_k)
\end{aligned}
$$

The first construct defines a *mixin class* (which may be abstract). A mixin may be defined by inheritance from another mixin class, and implement several interfaces, just as with ordinary classes. A mixin class will typically also *expect* a superclass providing a particular interface; the mixin class uses the `inner` construct to refer to methods and constructors in the superclass. The second mixin definition construct, *mixin combination*, is the interesting construct. It allows several mixins to be combined into a single mixin class; this combined class has the union of all of the fields of the original class. Where two mixin classes provide a definition for the same method, the definition in the left class overrides the definition in the right class. Methods and constructors in the left class can refer to method and constructor definitions in the right class through the `inner` keyword. The code sequence:

```
abstract mixin IM { abstract int getX(); }
mixin M1 expects IM { int getX() { return inner.getX(); } };
mixin M2 extends IM { int x; int getX() { return x; } };
mixin M3 combines M1, M2;
```

is analogous to the conventional class definitions:

```
class C2 { int x; int getX() { return x; } };
class C3 extends C2 { int getX() { return super.getX(); } };
```

The strength of mixin-based inheritance is that the extending class can be defined independently of the class which it extends. `M1` in this example is an *abstract subclass* that can be used to extend any class with a `getX` method.

We allow nested classes (including nested mixin classes) in mixin classes. The semantics of mixin combination with nested classes are different from the semantics for inheritance of ordinary Java classes with nested classes. Mixin combination must be recursively applied to similarly-named nested mixin classes, as part of composition. Where there are similarly-named nested classes or interfaces in two mixins being combined, we require that the definitions of these classes or interfaces come from the same “base” mixin class from which each mixin inherits, ensuring that similarly named classes and interfaces nested in different mixins have the same definition. This is sufficient for the examples in this paper.

The initial Interp class can now be defined as an *abstract mixin class*:

```
public abstract mixin Interp {
   public interface Expr {
      public Computation eval ();
   };
   public interface Computation {
      public ValueOutputContext run (@ThisInputContext ic);
   };

   static public mixin InputContext as ThisInputContext { };
   static public mixin OutputContext as ThisOutputContext { };
```

In this abstract mixin the input and output contexts are empty; we expect concrete mixins that extend this to extend the input and output contexts as appropriate. Therefore the input to a computation has type ThisInputContext, a self type that refers to the final input context type (after extensions). In contrast to the "pure" Java declaration in Sect. 2, the argument to run is not of type Interp.InputContext, but carries the more precise type of whatever final extension of InputContext is used in the combination of a collection of mixins that inherit from Interp. This avoids the use of runtime type tags and downcasting to interrogate an input context for a required field (as in the pure Java code). So one benefit of having explicit support for typed mixins is that uses of input and output contexts can be statically type-checked, providing greater reliability. Another potential benefit is that compilation techniques can allow fields of input and output contexts to be accessed in constant time, avoiding the runtime lookup used in the Java code in Sect. 3.

Similarly the output from a computation has type ThisOutputContext.

```
  public interface Value { };
  static public final class ValueOutputContext {
    public Value v;
    public @ThisOutputContext oc;
    public ValueOutputContext (Value v, @ThisOutputContext oc) {
      this.v = v; this.oc = oc;
    }
  }
```

The unit and compose operations are now defined as *static abstract methods*. Implementations of these methods are provided by subclasses of Interp. Mixin combination merges the definitions of these methods in the final interpreter class:

```
  public static abstract Computation unit (Value v);

  public interface Continuation {
    public Computation kontinue (Value v);
  };
  public static abstract Computation compose
         (Computation left, Continuation right);
```

As with the pure Java code, these operations rely on other methods to provide the glue for generating output and input contexts. These methods are also defined as static abstract methods, with the intention that they are instantiated by inheritance and these instantiations are then combined using mixin combination:

```
  public static abstract @ThisOutputContext outputContext
         (Value v, @ThisInputContext ic);
  public static abstract @ThisInputContext leftInputContext
         (Computation right, @ThisInputContext ic);
  public static abstract @ThisInputContext rightInputContext
         (@ThisInputContext ic, @ThisOutputContext leftOutputCtxt);
};
```

A default definition of these methods is provided in Fig. 2. We have the following correspondence between expressions in Fig. 1 and in Fig. 2:

Pure Java	Mixin Java
root.outputContext	outputContext
root.leftInputContext	leftInputContext
root.rightInputContext	rightInputContext
root.inputContextFactory	new ThisInputContext ()
root.outputContextFactory	new ThisOutputContext ()

The Function mixin is now defined as a concrete mixin that inherits from the Interp abstract mixin, providing implementations for the abstract static methods. This mixin expects to be combined with another mixin that also inherits from Interp. This is in contrast with Defaults, which does not use inner and therefore does not provide an expectation clause, since it does not expect to be combined with another mixin providing some method definitions.

The inner construct is used to patch together implementations of methods contributed by different mixins in a mixin combination. Occurring in a method, it denotes the code for that method contributed by the next mixin in a mixin combination. Occurring in a constructor, it denotes the code for that constructor in the next mixin. For example:

```
public mixin Function extends Interp expects Interp {

  public mixin InputContext extends Interp.InputContext {
     public Env env;
     public InputContext () { inner(); this.env = new Env(); }
  };

  public static @ThisInputContext leftInputContext
         (Computation right, @ThisInputContext ic) {
     @ThisInputContext left = inner.leftInputContext (left, ic);
     left.env = ic.env;
     return left;
  };
  public static @ThisInputContext rightInputContext
         (@ThisInputContext ic, @ThisOutputContext leftOutputCtxt) {
     @ThisInputContext right =
         inner.rightInputContext (ic, leftOutputCtxt);
     right.env = ic.env;
     return right;
  };

  public interface Closure extends Value {
     public Computation exec (Expr arg);
  };
```

```
public mixin Defaults extends Interp {

  public static Computation unit (final Value v) {
    return new Computation () {
      public ValueOutputContext run (@ThisInputContext ic) {
        return new ValueOutputContext (v, outputContext (v, ic));
      }
    }
  };

  public static Computation compose (final Computation left,
                                     final Continuation right) {
    return new Computation () {
      public ValueOutputContext run (@ThisInputContext ic) {
        ValueOutputContext voc = left.run (leftInputContext (right, ic));
        return right.kontinue(voc.v).run (rightInputContext (ic, voc.oc));
      }
    }
  };

  public static @ThisOutputContext outputContext
         (Value v, @ThisInputContext ic) {
    return new ThisOutputContext ();
  };
  public static @ThisInputContext leftInputContext
         (Computation right, @ThisInputContext ic) {
    return new ThisInputContext ();
  };
  public static @ThisInputContext rightInputContext
         (ThisInputContext ic, @ThisOutputContext leftOutputCtxt) {
    return new ThisInputContext ();
  };

};
```

Fig. 2. Default Definitions in Mixin Java

In this case, `leftInputContext` and `rightInputContext` use `inner` to refer to extensions of these methods contributed by other mixins, while the nullary constructor for the `InputContext` nested mixin uses `inner` to refer to the definition of the nullary constructor contributed by other mixins. In this example `inner` is analogous to `call-next-method` in CLOS method combination [26]. It has some relationship to `inner` in Beta [32] and `super` in Java [25]. We provide `inner` in addition to `super` to refer to method definitions in combining classes, because `super` is independently useful in a mixin when referring to the base mixin from which a mixin inherits (e.g., `Function` inheriting from `Interp`).

The use of precise mutually-recursive self types means that dynamic type casts (and potentially dynamic type-based searching) are unnecessary in accessing fields from contexts. For example:

```
public class Var implements Expr {
  public final String x;
  public Computation eval () {
    return new Computation () {
      ValueOutputContext run (@ThisInputContext ic) {
        Computation comp = unit (ic.env.lookup(x));
        return comp.run (ic);
      }
    };
  }
};
```

If the modular interpreters are defined as mixin classes in this fashion, then the final interpreter class can be obtained by combining mixin classes:

```
mixin Interp combines Control, Function, Imperative, Defaults;
```

7 Related Work

As interest in DSLs has increased in recent years, others have also investigated approaches to providing modular reusable interpreters for DSLs. Steele [41] considered how to "plug" together modular interpreters written in a monadic style. His approach requires every interpreter to have an environment argument, and also relies on a fair amount of run-time type coercions. Moggi [35], Liang et al [30], and Espinosa [20] all independently suggested combining monadic definitions using *monad transformers*. Wansbrough and Hamer [43] combine the approach of monad transformers with action semantics [37]. We have the following comparison of the approaches:

Approach	Monad Transformer	Extensible Computation
Interpreter Definition	Data Types and Type Classes (or Parameterized Modules)	Mixin Class
Interpreter Composition	Overload Resolution (or Generic Instantiation)	Mixin-Based Inheritance

Monad transformers require a fairly complicated coding style, instantiating many parameterized definitions. Some of this complication can be hidden using Haskell type classes. However type classes are themselves a fairly complicated construct. For the flavor of type classes required for defining monad transformers (multi-parameter constructor type classes), there are still some open issues to be resolved. Also the approach of monad transformers places a heavy reliance on compiler optimizations to obtain reasonable implementations, whereas our approach relies only on an implementation of mixin-based inheritance.

Filinski [22] takes another approach with "layering" monadic definitions. Rather than defining modular interpreters in a metalanguage, his interest is in defining extensions of a base language in a monadic style and then reflecting those extensions into the base language. For efficient implementation he relies on imperative state and a `call-with-current-continuation` primitive [11] in the base language. Filinski criticizes work on monad transformers as being too ambitious. For example monad transformers allow non-determinism to be added to an interpreter by layering a monad transformer that enriches a computation with sets of values. This is a fairly abstract and high-level representation of non-determinism. Filinski instead relies on using a lower-level resumption semantics to implement pre-emptive threads.

Our approach lies somewhere between Filinski's approach and that of Liang et al. We do not allow the general transformations on monads provided by Liang et al. On the other hand, our more limited approach may be more readily usable since the "plugging together" code in our approach bears some relationship to semantic definitions in attribute grammars, a familiar and highly successful approach to providing static semantics. This conjecture can only be verified by suitable long-term experimentation.

Cartwright and Felleisen [9] give a protocol for structuring denotational semantics so that such a semantics can be incrementally extended, based on passing an "extension" function to a semantic interpreter. This latter extension function is similar to the `nextInterp` instance variable used in this paper. Krishnamurthi and Felleisen [29,28] develop a general theory for software reuse that is based on extensible datatypes and extensible functions over those datatypes. They use the protocol proposed by Cartwright and Felleisen to develop an extensible version of the Interpreter design pattern [24].

Lieberherr [31,33,39] advocates software reuse based on *adaptive object-oriented design*. This is a methodology for designed object-oriented components supporting mixin-based inheritance. Mixin-based inheritance is an ingredient of our approach to "plugging together" software systems. So our approach can be viewed as an instance of adaptive software reuse, with explicit consideration of how to achieve adaptive software reuse for a particular application domain (modular interpreters).

There has been work on the modular definition and combination of attribute grammars. Classic attribute grammars [27] already allow a modular decomposition of the static semantics of a programming language, allowing attributes and their defining semantic equations to be associated with particular phrase structures in the language. Modular attribute grammars [15] and composable attribute grammars [21] seek to decompose the semantic processing for each phrase structure into stages. This is not currently an issue with our work, but could be at some point in the future. Recent work looking at combining aspect-oriented computing and attribute grammars [14] is largely orthogonal to the work presented here, since the aim of aspect-oriented compilers is (as with modular and composable attribute grammars) to decompose the processing at each abstract syntax tree node into stages.

In the software engineering community, reuse of domain-specific programming languages has recently gained attention. The most prominent examples of this are the IP project from Microsoft Research [1] and the JTS project from the University of Texas [4]. The former approach is based on a programming environment for describing domain-specific abstractions, and then refining those abstractions to specific implementations. The JTS project provides support for reusing abstract syntax trees and operations on those trees, with an emphasis on "hygienic" syntactic manipulations of ASTs. An interesting point of comparison is that the JTS project recognizes the importance of mixin-based inheritance, at the granularity of collections of classes rather than individual classes, for reusing DSLs. For example, if a language syntax has nonterminals for expressions and statements, then mixin-based inheritance for language fragments must be able to simultaneously inherit from both expression and statement classes. This is certainly a part of our framework; however our framework goes beyond this, to considering inheritance for semantic definitions and the proper composition mechanisms that allow modular semantic definitions to be combined via inheritance.

8 Conclusions

The current work should be viewed as an initial experiment to demonstrate the viability of the approach. An evaluator that takes an environment, store and continuation as arguments is a paradigmatic example from programming language semantics. We expect that most language semantics would be straightforward adaptations of these fragments. For example, a language with backtracking can be obtained by threading both success and failure continuations through the semantics, in an analogous manner to the continuation-passing interpreter. A language with persistence and state undo can be done using persistent stores, a straightforward generalization of the interpreter with state. Combining these two languages provides a language with Prolog-like semantics.

The semantics of inheritance is now well-understood as the incremental extension of the fixed points of recursive definitions [12,13]. The incremental extension of programming language implementations is therefore a useful exercise for any inheritance mechanisms, because of the highly recursive nature of programming language syntax. We have examined the mechanisms that might be added to Java to support this style of programming in a type-safe and efficient manner. These extensions constitute the combination of mixin-based inheritance and mutually recursive self types. Although each of these has been examined in isolation, there has been limited consideration of their combination [7,17,2,18,19]. The current work provides some motivation for the further examination of the semantics of this combination in Java.

References

1. W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi. Transformation in intentional programming. In *Proceedings of the International Conference on Software Reuse*. ACM Press, 1998.
2. David Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998.
3. Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam: A smooth extension of Java with mixins. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Cannes, France, June 2000. Springer-Verlag.
4. Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *Proceedings of the International Conference on Software Reuse*. ACM Press, 1998.
5. Edoardo Biagioni, Robert Harper, Peter Lee, and Brian G. Milnes. Signatures for a network protocol stack: A systems application of standard ML. In *Proceedings of ACM Symposium on Lisp and Functional Programming*, pages 55–64, Orlando, Florida, January 1994. ACM Press.
6. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, pages 303–311. ACM Press, October 1990. SIGPLAN Notices, volume 25, number 10.
7. Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *International Conference on Computer Languages*, pages 282–290. IEEE, 1992.
8. Kim Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
9. Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Science*, volume 789 of *Lecture Notes in Computer Science*, pages 244–272. Springer-Verlag, 1994.

10. J. Craig Cleveland. Building application generators. *IEEE Software*, July 1988.
11. William Clinger and Jonathan Rees, editors. *IEEE Standard for the Scheme Programming Language*. Institute for Electrical and Electronic Engineers, 1991. IEEE Standard 1178-1990.
12. William Cook. *A Denotational Semantics for Inheritance*. PhD thesis, Brown University, 1989.
13. William Cook and Jens Palsberg. A denotational semantics for inheritance and its correctness. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1989.
14. Oege de Moor, Simon Peyton-Jones, and Eric Van Wik. Aspect-oriented compilers. In *Proceedings of the First International Symposium on Generative and Component-Based Software Engineering*, Erfurt, Germany, September 1999.
15. G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *Computer Journal*, 33(3):164–172, 1990.
16. Dominic Duggan. Modular domain-specific languages: A mixin module-based approach. Submitted for publication, 1999.
17. Dominic Duggan and Constantinos Sourelis. Mixin modules. In *Proceedings of ACM International Conference on Functional Programming*, pages 262–273, 1996.
18. Dominic Duggan and Constantinos Sourelis. Mixin-based, module-based inheritance. Technical Report 2001, Stevens Institute of Technology, Hoboken, New Jersey, 2000.
19. Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.
20. David Espinosa. *Semantic LEGO*. PhD thesis, Columbia University, May 1995.
21. R. Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: support for modularity in translator design and implementation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 223–233, 1992.
22. Andrej Filinski. Representing layered monads. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 175–188, 1999.
23. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, January 1998.
24. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.
25. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1997.
26. S. C. Keene. *Object Oriented Programming in Common Lisp: A Programming Guide in CLOS*. Addison-Wesley, 1989.
27. D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, February 1968. Correction, **5**, 1, 95–96, March 1971.
28. Shriram Krishnamurthi and Matthias Felleisen. Toward a formal theory of extensible software. In *Proceedings of ACM Symposium on Foundations of Computer Science*, 1998.
29. Shriram Krishnamurthi, Matthias Felleisen, and Daniel Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113, 1998.
30. Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 333–343, San Francisco, January 1995. ACM Press.
31. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
32. Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. ACM Press, 1993.
33. Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1998.

34. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Revised Definition of Standard ML.* The MIT Press, 1997.
35. Eugenio Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer, University of Edinburgh, Scotland, 1990.
36. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
37. Peter Mosses. *Action Semantics*. Number 26 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
38. J. Neighbours. Draco: A method for engineering reusable software components. In T. J. Biggerstaff and A. Perlis, editors, *Software Reusability*. Addison-Wesley/ACM Press, 1989.
39. Johan Ovlinger and Mitchell Wand. A language for specifying traversals of object structures. In *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, 1999.
40. R. Prieto-Diaz and G. Arango. *Domain Analysis and Software Systems Modelling*. IEEE Computer Society Press, 1991.
41. Guy Steele. Building interpreters by composing monads. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 472–492. ACM Press, 1994.
42. P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.
43. Keith Wansbrough and John Hamer. Modular monadic action semantics. In *USENIX Symposium on Domain-Specific Languages*, pages 157–170, Santa Barbara, California, October 1997. USENIX.

Generic Wrappers

Martin Büchi[1] and Wolfgang Weck[2]

[1] Turku Centre for Computer Science, Åbo Akademi University,
Lemminkäisenkatu 14A, FIN-20520 Turku,
Martin.Buechi@abo.fi
[2] Oberon microsystems Inc., Technoparkstrasse 1,
CH-8005 Zürich,
weck@oberon.ch

Abstract. Component software means reuse and separate marketing of pre-manufactured binary components. This requires components from different vendors to be composed very late, possibly by end users at run time as in compound-document frameworks.

To this aim, we propose generic wrappers, a new language construct for strongly-typed class-based languages. With generic wrappers, objects can be aggregated at run time. The aggregate belongs to a subtype of the actual type of the wrapped object. A lower bound for the type of the wrapped object is fixed at compile time. Generic wrappers are type safe and support modular reasoning.

This feature combination is required for true component software but not achieved by known wrapping and combination techniques, such as the wrapper pattern or mix-ins.

We analyze the design space for generic wrappers, e.g. overriding, forwarding vs. delegation, and snappy binding of the wrapped object. As a proof of concept, we add generic wrappers to Java and report on a mechanized type soundness proof of the latter.

1 Introduction

Component software enables the development of different parts of large software systems by separate teams, the replacement of individual software parts that evolve at different speeds without changing or reanalyzing other parts, and the marketing of independently developed building blocks. Components are binary units of independent production, acquisition, and deployment [30].

Component technology aims for late composition, possibly by the end user. Compound documents, e.g. a Word document with an embedded Excel spreadsheet and a Quicktime movie, as well as Web browser plug-ins and applets are examples of this. Late composition is a major difference between modern components and traditional subroutine libraries, such as Fortran numerical packages, which are statically linked by the developer.

Flexible late composition is one goal, prevention of unsafe compositions leading to system failures is the other. Static safety assertions are especially important for software components that are composed by third parties because systematic integration testing by the component developer is practically impossible. Where compile-time checks are

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 201–225, 2000.
© Springer-Verlag Berlin Heidelberg 2000

impossible, as early as possible run-time detection of errors facilitates systematic testing and debugging. Type systems can make certain safety guarantees, but they often do so at the price of reduced flexibility.

In this paper we present an inflexibility problem in class-based languages and propose a new solution that partly borrows from prototype-based languages yet retains the possibility for maximal static and as-early-as-possible run-time error detection and modular reasoning.

Late composition is most pressing for items defined by different components, which may themselves be combined by an independent assembler or even by the user at runtime. Component standards such as Microsoft's COM [25], JavaBeans [29], and CORBA Components [21] are on the binary level. Components can be created in any language for which a mapping to the binary standards exists. However, binary standards are most easily programmed to in languages that support the same composition mechanisms. Furthermore, only direct language-level support can provide the desired machine checkable safety using types. Hence, composition mechanisms in programming languages are relevant, even though components are binary units.

The mechanism suggested in this paper is partly inspired by COM's aggregation, but it doesn't yet have an exact equivalent in any of the aforementioned binary component standards.

Overview. Section 2 illustrates with examples a problem of existing composition mechanisms and defines the requirements for a better solution. In Sect. 3, we show why existing technology does not sufficiently address these requirements. We introduce generic wrappers as a solution to the aforementioned problems in Sect. 4. Next, we discuss the design space for generic wrappers in Sect. 5 and the interplay with other type mechanisms in Sect. 6. As a proof of concept we add generic wrapping to Java in Sect. 7 and report on a mechanized type soundness proof of the extended language in Sect. 8. Finally, Sect. 9 points to related work and Sect. 10 draws the conclusions.

2 The Problem

In this section, we describe some applications that cannot be satisfactorily realized with existing composition mechanisms. We also introduce some terminology, and distill a set of requirements.

2.1 Examples

We illustrate the problem with examples in the realm of compound documents. Embedded views in compound documents for on-screen viewing, such as an Excel spreadsheet in a Word document, may be so large that they require their own scroll bars. Likewise, the user may want to add borders or identification tags to embedded views. It is even possible, that a user wants several such decorators added to the same embedded view.

There may exist different scroll bars from different vendors, which don't know all the other decorator or embedded view vendors. Decorators are typical examples of third-party components that users want to select to meet their specific needs. One user may want proportional scroll bars, another may like blinking borders to draw the boss'

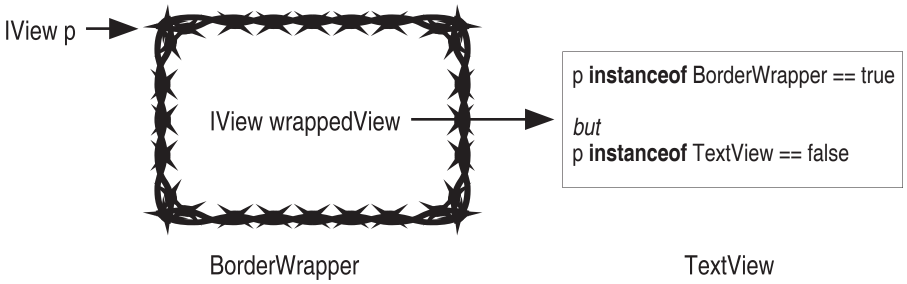

IView p
IView wrappedView
p instanceof BorderWrapper == true
but
p instanceof TextView == false
BorderWrapper
TextView

Fig. 1. The wrapper is not fully transparent to clients of the embedded view

attention to the excellent sales figures, and still another may require immutable 128-bit identification tags.

In a compound-document framework similar to Java Swing or Microsoft OLE, let IView be the interface implemented by all classes whose instances can be displayed on screen and inserted into containers. Typical examples of classes implementing IView are TextView, GraphicsView, SpreadsheetView, and ButtonView.

One way to implement decorators is with wrappers [9, Decorator Pattern]. A border wrapper is itself a view, that is it implements the IView interface. Hence it can itself be inserted into a compound document container. Furthermore the wrapper contains a reference of type IView to a wrapped view, which in a specific instance may be a TextView. The wrapper forwards most requests to the wrapped view, possibly after performing additional operations such as drawing the border.

Unfortunately, this approach has a serious disadvantage. If we wrap a border around a TextView, then the aggregate is only a BorderWrapper, but not a TextView with all of the latter's methods (Fig. 1). Hence, a spell check operation on all embedded text views in a document will not recognize a bordered TextView as containing text, unless it knows how to search inside wrappers from different manufacturers.

A standard interface, like IViewWrapper to be implemented by all view wrappers could ease the problem of searching inside different wrappers:

```
interface IViewWrapper {
  IView getWrappee();
}
```

However, instead of a simple type test, the spell checker would have to loop through all the wrappers:

```
IView q=p;
while(!(q instanceof TextView) && q instanceof IViewWrapper) {
  q=((IViewWrapper)q).getWrappee();
}
if(q instanceof TextView) {...;}
```

This solution is cumbersome for several reasons: First, it requires 5 lines of code instead of a simple type test. Second, it only works if there is a unique standard for wrappers, such as IViewWrapper. Third, it doesn't let the wrapper maintain invariants ranging over both itself and the wrapped object because clients have direct access to the latter.

Support for certain common kinds of wrappers may also be built into the wrapped objects. For example, JComponent, the correspondence to our IView in Java Swing, supports borders as insets. However, identification tags and other kinds of wrapper that were not previewed by the Swing designers are left out.

As a second example, let us consider a forms container that requires all its embedded views to implement the interface IControl. Assume that ButtonView implements IControl and that BorderWrapper doesn't. Hence, a bordered ButtonView cannot be inserted into a forms container: The type system rightfully prevents us from passing a BorderWrapper wrapping a ButtonView as the first parameter to the method insert(IControl c, Point pos). Passing just the wrapped ButtonView as parameter to insert is not a solution, because we would loose the border. The only workaround is to change the type of the first parameter of insert to IView and test that the actual parameter implements IControl or wraps an object that does so.

2.2 Terminology

We use the following terminology: A wrapped object is called a *wrappee*. A wrapper and a wrappee together are referred to as an *aggregate*. The declared type of a variable is referred to as *static* (compile-time) *type*. The type of the actually referenced object is called the variable's *dynamic* (actual, run-time) *type*. Likewise, we distinguish between the *static* (declared, compile-time) and the *dynamic* (actual, run-time) *wrappee type*. For example, for an instance of BorderWrapper, declared to wrap an IView, and actually wrapping a TextView, the static wrappee type is IView and the dynamic wrappee type is TextView.

In discussions, we use the notation C.m to refer to the implementation of instance method m in class C. The subtype relation is taken to be reflexive; e.g., TextView is a subtype of itself.

Except where otherwise stated, the discussion in the first 6 sections applies to most strongly-typed class-based languages such as Java [10], Eiffel [17], and C++ [28]. For simplicity, we use Java terminology throughout the paper. A Java interface corresponds to a fully abstract class in Eiffel and C++.

2.3 Requirements

From the above examples we can distill a number of requirements for a wrapping mechanism. Numbers in parentheses refer to the summary of requirements in Fig. 2.

The user wants to select which border to wrap around which view. At compile time, the implementor of BorderWrapper doesn't know whether an instance of her class will wrap a TextView, a GraphicsView, or any other view that might even be only implemented in the future. Thus, the actual type and instance of the wrappee must be decidable at run time (1). Furthermore, wrappers must be applicable to any subtype of the static wrappee type (2).

An aggregate of a BorderWrapper wrapping a ButtonView should be insertable into a controls container, even though only the wrappee implements the interface IControl.

1. *Run-time applicability.* The actual type and instance of the wrappee must be decidable at run time.
2. *Genericity.* Wrappers must be applicable to any subtype of the static wrappee type.
3. *Transparency.* An aggregate should be an element of the wrapper and the actual wrappee type.
4. *Overriding.* Wrappers must be able to override methods of the wrappee.
5. *Shielding.* A wrapper should be able to control whether clients can directly access the wrappee.
6. *Safety.* The type system should prevent as many run-time errors as possible statically and signal errors as early as possible at run time.
7. *Modular reasoning.* Modular reasoning should be possible in the presence of wrapping.

Fig. 2. Requirements for a Wrapping Mechanism

Therefore, an aggregate should be an element of the actual wrappee type (3). This also implies that all methods of the wrappee can be called by clients and that they can make these calls directly on a reference to the wrapper.

Upon calling paint on an aggregate of a BorderWrapper and a TextView, the border's paint method should be executed. The latter first draws the border and then calls the paint method of the wrapped view with an adapted graphics context. Thus, wrappers must be able to override methods of the wrappee (4).

If clients can have direct references to the wrappee, they can call overridden methods. For example, a client could call the paint method of the embedded view with the graphics context (dimensions) of the whole aggregate. Hence, a wrapper should be able to control whether clients can directly access the wrappee (5).

Early detection of errors has already been identified as general requirements for component-oriented programming. We restate it here as an explicit requirement (6) for the purpose of assessing composition mechanisms.

The possibility for modular (component-wise) reasoning is another key requirement (7) for any mechanism targeted at component-based programming because of the independent development of components [30].

Finally, it is desirable that classes are not required to follow any coding standards for their instances to be wrappable. Otherwise, instances of classes programmed to different standards and of legacy classes are left out. Since certain coding standards can be established, as shown by JavaBeans, and since certain automatic rewriting —even of binary code— is possible, we consider this as a nice-to-have feature, but do not make it a formal requirement.

3 Why Existing Technology Is Insufficient

In this section we show why existing technology fails to address the above requirements.

Inheritance. Feature combination by multiple inheritance creates specialized combination classes, such as BorderedTextView and BorderedGraphicsView. Thus, it combines the functionality of the wrapper and the wrappee into a single object. However, combinations can only be made at compile time by a vendor having access to both the border

Requirement Technology	Run-time applicability (1)	Genericity (2)	Transparency (3)	Overriding (4)	Shielding (5)	Safety (6)	Modular reasoning (7)
Inheritance		(b)			n/a		(c)
Parameterized mix-ins		(b)			n/a		(c)
Containment		(b)		(d)	(d)	(e)	
Specialized wrappers[(a)]	(f)	(b)					
Bottleneck interface							
Dual interface		(b)				(e)	
Delegation in prototype-based languages			n/a				

(a) If only used with specific type, otherwise like containment.
(b) Yes, but with exceptions due to signature clashes.
(c) Limited due to tight coupling.
(d) Either full functionality availability or overriding and shielding.
(e) Type safety for static wrappee type.
(f) Type determined at compile time.

Fig. 3. Properties of Existing Technologies

and the view. Run-time feature composition, e.g., in compound documents, is impossible with inheritance. Hence, inheritance fails requirement (1). The modular reasoning requirement (7) is not fully satisfied because of the close coupling between super- and subclass, leading to the semantic fragile base class problem [19].

Mix-ins make it easier to create different combinations, even in single inheritance languages. However, combinations must be made at compile time. Thus, mix-ins also fail the requirement of run-time applicability (1).[1]

Containment. The containment approach, also known as the decorator pattern [9], has already been sketched along with the presentation of the example in Sect. 2.1. It's main problem is that the aggregate is not a subtype of the actual wrappee. Thus, it fails the transparency requirement (3).

Specialized wrappers can be created at compile time for specific known wrappee types. However, specialized wrappers aren't of much help in a component market with mutually unaware vendors.

Bottleneck and dual interfaces, i.e., a single message handler method, are variations of the containment approach. They make the full functionality of the actual wrappee available through the wrapper, but lose the benefits of the static type system. Thus, they fail both the safety (6) and the transparency (3) requirements.

Delegation in prototype-based languages. Prototype-based languages, such as Self [31], use a parent object to which the receiving object delegates messages that it does not understand itself. A bordered text view could be implemented by a border object with a text view parent. Due to the lack of (static) typing and because of the possibility to reassign the parent object at any time, prototype-based languages fail the requirements of safety (6) and modular reasoning (7) [9].

[1] Most genericity mechanisms, such as GJ, do not support mix-ins because they do not allow the type parameter to be used as a supertype of the parameterized type.

Summary. We conclude that none of the existing technologies gives a satisfactory solution to the problem at hand. Figure 3 summarizes the results. Less common solution approaches are described in Sect. 9. More detailed refutations appear in [4].

4 Generic Wrappers

To solve the problem stated in Sect. 2, we introduce generic wrappers. Generic wrappers are classes that are declared to wrap instances of a given reference type (class, interface) or of a subtype thereof. Like an extends clause to specify a superclass, we use a wraps clause to state the static wrappee type. This also declares the wrapper class to be a subtype of the static wrappee type. For example, the declaration

```
class LabelWrapper wraps IView {...}
```

states that each instance of the class LabelWrapper wraps an instance of a class that implements IView. The declaration makes class LabelWrapper a subtype of IView. Thus, instances of LabelWrapper can be assigned to variables of type IView and LabelWrapper has all public members (methods, fields) of IView.

To assure that this subtyping relationship always holds (and thereby that forwarding of calls never fails) must instances of LabelWrapper always wrap an instance of a subtype of IView —already during the execution of constructors. Hence, the wrappee must be passed as a special argument (in our syntax delimited by <>) to class instance creation expressions:

```
TextView t = ...; IView v = new LabelWrapper<t>(...);
```

The compiler checks that the declared type of variable t is a subtype of the static wrappee type. The wrapper class instance creation expression throws an exception if the value of t is null or if t were an expression and its evaluation throws an exception. In both cases, no wrapper object is created and the value of v remains unchanged.

The particularity of generic wrappers is that their instances are not only of the static, but also of the actual wrappee type. For example, a LabelWrapper wrapping a TextView is also of the latter type and not just of type IView. Hence, such an aggregate can be assigned to a variable of type TextView and the latter's methods can be called on it. In the following program fragment, which is based on the definition of LabelWrapper above, the type test returns true and the cast succeeds:

```
IView v = new LabelWrapper<new TextView()>(...);
TextView t2; if(v instanceof TextView) {t2=(TextView)v;}
```

Methods declared in the wrapper override those in the wrappee analogously to overriding in subclasses.

In constructors and instance methods of generic wrappers, the keyword wrappee references the wrappee. It can be treated like an implicitly declared and initialized final instance field. Hence, wrappers can call overridden methods of the wrappee using the keyword wrappee corresponding to super for overridden methods of superclasses. For example, the paint method of BorderWrapper might look as follows:

```
public void paint(Graphics g) {
   ...; // paint border
   wrappee.paint(g1); // paint wrapped view with adapted graphics context
}
```

Preliminary evaluation. Although this basic definition still leaves many aspect open, we can evaluate which requirements (Fig. 2) it fulfills independently of how the details are fixed. The actual type and instance of the wrappee can be decided upon at run time. Hence, requirement (1) is satisfied.

Wrappers are applicable to a all of the static wrappee type's subtypes, for which no unsound overriding would occur. Thus, the genericity requirement (2) is mostly fulfilled.

As defined above, instances of generic wrappers are members of the actual wrappee type. Therefore, the transparency requirement (3) is satisfied.

The fulfillment of the shielding (5) and modular reasoning (7) requirements cannot be judged without fixing more details.

The compiler ensures that an aggregate is always of the static wrappee type and, thereby, that all calls to methods of the static wrappee type will succeed. Run-time tests can be used to check whether the aggregate is of a certain type. Only insufficiently guarded casts may fail. Calls to methods of the actual wrappee type always find a matching method. Hence, the type system fulfills the safety requirement (6) by preventing as many run-time errors as possible statically and signaling errors as early as possible at run time.

5 Design Space for Generic Wrappers

The basic definition of generic wrappers in the previous section leaves many aspects open. In this section, we investigate the design space for generic wrappers.

The time of binding has a major influence on the design space of generic wrappers as compared to inheritance. With inheritance, the superclass is bound at compile time. With generic wrappers the actual type and instance of the wrappee first become known at wrap time, that is, run time. Later binding brings flexibility, but means that certain compatibility checks asserting type soundness and, thereby, the success of all method lookups have to be delayed (Fig. 4). A notable feature of generic wrappers is that an existing wrapper object can be wrapped again. Thus, it remains always possible to add new functionality to an aggregate.

Dynamic linking partly blurs this distinction. The name of the superclass is fixed at compile time, but the actual version and, therefore, the members and their semantics are not known until load time. For example in Java, the loading of a class may be delayed until an instance thereof is created. In this case, the compatibility with the used superclass is checked as late as the compatibility between a wrapper and the actual wrappee type. Thus, dynamic linking postpones compatibility checking to run time without fully exploiting the flexibility thereof.

5.1 Overriding of Instance Methods

Overriding of instance methods in subclasses is governed by certain rules to guarantee both type and semantic soundness. The same rules extend to overriding of methods

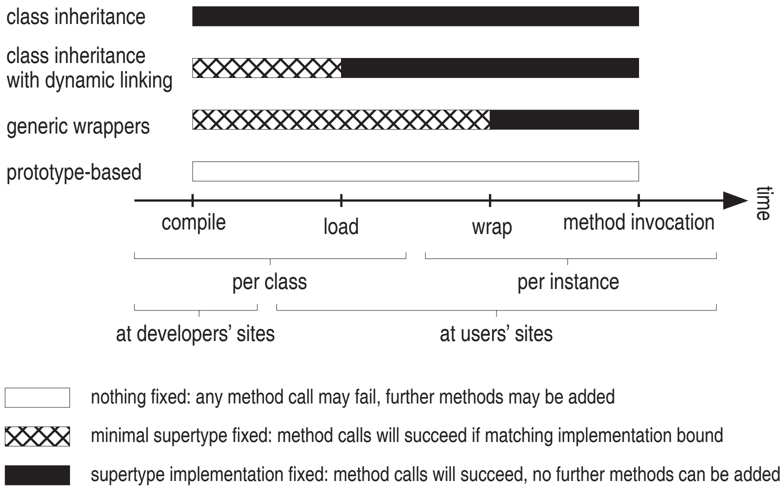

class inheritance
class inheritance with dynamic linking
generic wrappers
prototype-based
time
compile
load
wrap
method invocation
per class
per instance
at developers' sites
at users' sites
nothing fixed: any method call may fail, further methods may be added
minimal supertype fixed: method calls will succeed if matching implementation bound
supertype implementation fixed: method calls will succeed, no further methods can be added

Fig. 4. What Is Asserted to Hold from Where on?

of the wrappee by methods of the wrapper. For example to guarantee type soundness in Java, the overridden method must not be final, the return type of the overriding method must be the same as that of the overridden method, the overriding method must be at least as accessible, the overriding method may not allow additional types of exceptions to be thrown, and an instance method may not override a class method. To also guarantee semantic soundness, the overriding method must be a behavioral refinement of the overridden method [1].

Although the actual type of the wrappee isn't known until wrap time, we can perform certain checks at compile time. We can check that overriding of methods of the static wrappee type by methods of the wrapper respect the above rules. Any violation of the type rules would necessarily also lead to a violation in combination with any actual wrappee type, i.e., a subtype of the static wrappee type.

Because the actual wrappee may have more methods than the static wrappee type, overriding conflicts may nevertheless occur at wrap time, i.e., when the combination of the wrapper and the wrappee first becomes visible. In Fig. 5, three overriding conflicts occur when wrapping an instance of A in an AWrapper. The methods A.m an A.o would be overridden by semantically incompatible ones and AWrapper.n cannot override A.n because they have different return types.

Below we discuss two approaches to this problem. The first checks type soundness at wrap time and throws an exception if wrapping would be type unsound. To decrease the probability of unsound overriding, we suggest a number of coding conventions. The second approach avoids wrap-time type problems by relying on a different form of method lookup and subsumption. We conclude with a short refutation of static approaches.

In this section we assume that there are no final classes and no method header specialization in subtypes (overriding non-final with final methods, overriding with restricted exception throws clauses and higher accessibility, as well as covariant return type and contravariant parameter type specialization) in our language. The interaction

```
interface IA {
  int m(); // return 0 or 1
}

class AWrapper wraps IA {
  public int m() {return 0;};
  public void n() {...};
  public int o() {return 0;};
}
```

```
class A implements IA {
  public int m() {return 1};
  public int n() {...};
  public int o() {return 1};
}

IA a=new A();
AWrapper w=new AWrapper<a>();
```

Fig. 5. Overriding Example

of final classes and method header specialization with generic wrappers is discussed in Sect. 6.2.

Wrap-time tests and coding conventions. At wrap time, we can automatically check whether overriding of methods of the actual wrappee by the wrapper is type sound. If this is the case, we can create the wrapper instance. Otherwise, we throw an exception. Wrap-time tests require enough information in the binary code. Java byte code, for example, satisfies this requirement.

Wrap-time exceptions are undesirable, yet they are preferable over unsuccessful method lookup as in prototype based languages like Self. First, if components are combined by an assembler, she can much more easily check all combinations than all method calls on all combinations. Second, if an error occurs, detecting it as early as possible facilitates debugging, as expressed by requirement (6).

To reduce the probability of wrap-time type conflicts, we could use laxer rules in analogy to Java's binary compatibility. However, laxer typing rules threaten semantic soundness, which must be the ultimate goal. Hence, we believe that the strict rules should be used for generic wrappers at wrap time also.

We suggest to adhere to the following four coding conventions, which can greatly reduce the possibility of both type and semantic conflicts at wrap time:

(a) Classes only define (non private) methods declared in implemented interfaces.
(b) No two interfaces, not related by extension, declare methods with the same signature.
(c) Interfaces have semantic specifications and methods in classes are semantic refinements of their correspondences in the implemented interfaces.
(d) Method calls are only made on variables of interface, but not class types.[2]

We analyze the conventions for method o of Fig. 5. Convention (a) implies that both AWrapper and A implement interfaces declaring a method o. Furthermore, (b) dictates that this must be the same interface, say IO. The idea of behavioral subtyping [1] is that interfaces have semantic specifications and that methods in subtypes are behavioral refinements of the corresponding methods in their supertype. Assuming that

[2] Self calls, which are of course also allowed, are discussed in Sect. 5.3.

both AWrapper.o and A.o are refinements of IO.o, we can deduce that both 0 and 1 are correct return values. Finally, condition (d) implies that a call x.o() may only be written for x of static type IO. In this case, the value 0 returned by the overriding method AWrapper.m meets our expectations.

If (a) or (b) is not adhered to, then a type conflict may occur as illustrated by method n of Fig. 5. If (c) is not adhered to, it could be that IO.o specifies the return value to be 1, which would not hold in the above case. Finally, if (d) is not respected, we could make a call x.o on a variable of type A. If x contained a reference to an AWrapper wrapping an A, we would get a return value of 0 although we expected 1.

Conventions (a) and (d) could easily be enforced by a programming language. Instead of (b) a language can require qualified notation for member access instead of merging namespaces of interfaces. Convention (c) requires semantic proofs and is, therefore, more difficult to check. These conventions also avoid semantic problems in the overriding in subclasses. Hence, they are implicitly advocated as good style for object-oriented programming [9,30].

In conclusion, wrap-time checking allows us to avoid type unsound overriding. Furthermore, adherence to some also otherwise beneficial coding conventions can greatly reduce the possibility of type or semantic unsound overriding.

An alternative form of method lookup. An alternative is to have the wrapper only override methods already present in the static wrappee type. In Fig. 5, this would mean that only A.m would be overridden by AWrapper.m.

Instead of overriding additional methods of the actual wrappee, in the example A.n and A.o, we allow an aggregate to contain multiple methods with the same signature and base the dispatch on run-time context information. In the simplest case, the dispatch is based on the static type of the receiver:

```
AWrapper w=new AWrapper<new A()>();
int i; i=w.o(); // executes AWrapper.o, i=0
A a=(A)w; i=a.o(); // executes A.o, i=1
```

In more general cases, the dispatch is not only based on static, but on run-time context information, i.e., an object's history of subsumptions. To illustrate this, assume that method o is declared in interface IO and that both AWrapper and A implement IO. In the following code fragment, added to the above, the static type of the receiver is in both cases IO, but different implementations are executed:

```
IO x;
x=w; i=x.o(); // executes AWrapper.o, i=0
x=a; i=x.o(); // executes A.o, i=1
```

The problem is that there are two occurrences of IO in the aggregate. Thus, we have to choose one for subsumption.[3] Multiple non-virtual inheritance in C++ has a similar semantics.

[3] In our case, we have already subsumed the aggregate to be of type AWrapper, respectively A. A true choice would be needed in the first line if w were of a subtype of both AWrapper and A, e.g., the compound type [AWrapper, A] [3].

In languages that do not support method header specialization (Sect. 6.2) or final classes, this form of method lookup avoids wrap-time exceptions. However, to also achieve semantic soundness, we still need to adhere to the above four coding conventions. Otherwise, we could execute a.m() in the fragment above and be surprised that we don't get 1 as result. The soundness problems caused by specialization could only be avoided by fully giving up overriding.

Method lookup and subsumption are more complex with this approach. Furthermore, adding this to a single-inheritance language with 'normal' method lookup and subsumption for inheritance, we end up with two different forms of method lookup and subsumption. The technicalities of this approach for the case of compile-time composition of mix-ins can be found in [8].

Refutation of static approaches. Here we briefly discuss why some approaches that avoid possible wrap-time conflicts at compile time and that are based on normal overriding have serious deficiencies.

Allowing the wrapper to only override methods of the static wrappee type, but not add additional methods would avoid the problem of unsound overriding of additional methods in the actual wrappee, e.g. A.n. However, not allowing additional methods in the wrapper would be a severe restriction, which would greatly reduce the usefulness of generic wrappers. Furthermore, this approach would fail in languages that support final classes or method header specialization (Sect. 6.2).

Negative type information [24] could express that subtypes of IA must not have a method, like n, that might be overridden in an unsound way. However, this would also mean that an aggregate of an AWrapper and a subtype of IA would not be of a subtype of IA. Furthermore, negative type information cannot be expressed in type systems of current languages.

Requiring the exact type of the wrappee to be known at compile time, a third approach, would contradict the requirement of run-time applicability (1).

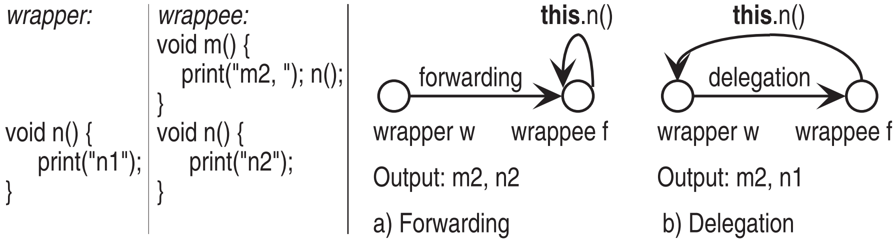
5.2 Hiding of Fields and Class Methods

In many languages, fields and class methods are hidden rather than overridden in subtypes. Hiding of fields, if permitted, is not problematic because the hiding field may have a different type than the hidden field. The static wrappee type is used to access hidden fields in the actual wrappee. Hiding of class methods is usually governed by similar requirements as overriding of instance methods. Thus, the same two options apply.

5.3 Forwarding vs. Delegation

The difference between forwarding and delegation is the binding of the self parameter in the wrappee when called through the wrapper. With delegation, the self parameter is bound to the wrapper, with forwarding it is bound to the wrappee. Figure 6 illustrates the difference with a client calling method m of the wrappee on a reference to the wrapper.

Forwarding is a form of automatic message resending; delegation is a form of inheritance with binding of the parent (superclass) at run time, rather than at compile/link time as with 'normal' inheritance [16].



```
wrapper:            wrappee:
                    void m() {
                       print("m2, "); n();
                    }
void n() {          void n() {
   print("n1");        print("n2");
}                   }
```

Fig. 6. Forwarding vs. Delegation

In all cases, super calls in the wrappee invoke methods of its superclass and not the wrapper's superclass. During these calls, this is bound to the wrappee with forwarding and to the wrapper with delegation.

Delegation has the advantage that the wrapper can better modify the behavior of the wrappee. With forwarding, the wrappee can better control its own behavior and avoid such problems as the semantic fragile base class problem. Generic wrapping as discussed in this paper works with both options. The reader is referred to [30,19,26] for further discussions of pros and cons.

5.4 Replacing a Wrappee

The wrappee of a given wrapper could be replaced by another object, the type of which is a subtype of the old dynamic wrappee type. It is not sufficient that the new wrappee is a subtype of the static wrappee type: A BorderWrapper wrapping a TextView can be referenced by a variable of static type TextView. Replacing the wrappee by a ButtonView would violate type soundness.

Although cyclic wrapping is type sound in combination with certain features, it is for semantic reasons mostly undesirable. Cyclic wrapping is prevented by the construction process, because the wrappee must be passed as an argument to the wrapper instance creation expression (Sect. 4). If we don't want cyclic wrapping, we also have to prevent it in the replacement of a wrappee.

For semantic reasons, we think that the wrappee should not be exchangeable. By fixing the wrappee for the lifespan of the wrapper, the system becomes more static and, therefore, simpler to analyze and reason about.

5.5 Direct Client References to the Wrappee

There are both advantages and disadvantages to allowing clients to hold direct references to the wrappee and being able to invoke the latter's methods —bypassing a possible overriding by the wrapper. On the positive side, this may give clients the possibility to invoke methods that are 'accidentally' overridden. For example, both BorderWrapper and TextView may define instance methods setColor with the same parameters. Without direct access to the wrappee, clients may not be able to change the text color. With direct access to the wrappee, this is possible. However, only clients that are aware that they reference a wrapped TextView rather than a bare one can do so. With the alternative method lookup, TextView.setColor can also be accessed through a cast, unless the method is already declared in the static wrappee type IView.

The disadvantage of direct client references to the wrapper is that clients may invalidate invariants ranging over both the wrapper and the wrappee. Furthermore, we end up with different reference values to the same aggregate; thus, loosing the unique identity and the possibility of direct reference comparison.

The transparency of generic wrappers reduces the need for direct client references. In the containment approach (Sect. 3) all functionality that the dynamic wrappee type provides beyond the static wrappee type can only be made accessible by giving clients direct access to the wrappee. With generic wrappers, on the other hand, the full functionality of the dynamic wrappee type is available through the wrapper.

Whether we allow the wrapper to hand out direct references or not, we have the problems of existing references to the wrappee and of the wrappee handing out self references. Even if we in principle permit direct references, we may want to restrict them to clients that explicitly ask for them and are aware of the dangers.

Redirection of existing references. The problem of existing references vanishes if there aren't any. In analogy to aggregation in Microsoft's COM, we could require the wrappee to be created along with the wrapper and not to allow the wrappee's constructor to pass out self references. However, experience with COM showed that this approach is often too restrictive [22].

The second best case is a single reference to the object to be wrapped. In type systems with aliasing control [12] that can guarantee uniqueness of references we could restrict wrapping to unique references. This single existing reference to the wrappee, which is used as argument in the wrapper construction, could then either be redirected to the wrapper or be set to null. The restriction to unique references may severely limit the applicability of wrappers. Furthermore, aliasing control is not common.

For mainstream languages we see the following options:

1. Keep the references to the wrapped object unchanged. This is only an option if we allow direct references to the wrappee. Unfortunately, clients won't recognize if an object they refer to has been wrapped. Hence, they might unknowingly invoke overridden methods of the wrappee and, thereby, cause the aforementioned semantic problems.
2. Update all existing references to point to the wrapper. Thanks to the transparency of generic wrappers this is sound. Since the type of a reference can only be increased by wrapping, assumptions that a reference is at least of a certain type are not falsified.

Handing out of self references. Wrappees may pass out self references, e.g. for event listener registration. If we don't want direct client references to the wrappee or only allow the wrapper to hand them out, we need to address this issue. The draconian solution is to disallow the use of this in the wrappee except for member access. This is, however, very restrictive and excludes instances of legacy classes not adhering to this restriction.

Alternatively, we may define this in the wrappee to reference the wrapper except when used for member access.

5.6 Multiple Wrapping

There are two forms of multiple wrapping, *conjunctive* and *disjunctive* (Fig. 7). Conjunctive (also called additive or recursive) wrapping applies multiple wrappers around each other. For example, we might wrap a TextView in a ScrollWrapper and the latter with a BorderWrapper.

Disjunctive wrapping presents the same wrappee with different wrappers. It has analogous drawbacks as direct client references to the wrappee, namely the possibility of invalidating invariants ranging over the wrappee and one of its other wrappers. With type transparency, disjunctive wrapping can in most cases be replaced by conjunctive wrapping because the full dynamic wrappee type is visible through all wrappers.

If we allow direct client references to the wrappee but not disjunctive wrapping, we have to define what happens if a client wraps an object that is already wrapped. The options are disallowing it and throwing an exception if tried, putting the new wrapper between the wrappee and the old wrapper, and applying the new wrapper around the old wrappee. Thanks to the transparency of generic wrappers, all options are type sound.

5.7 Concealment

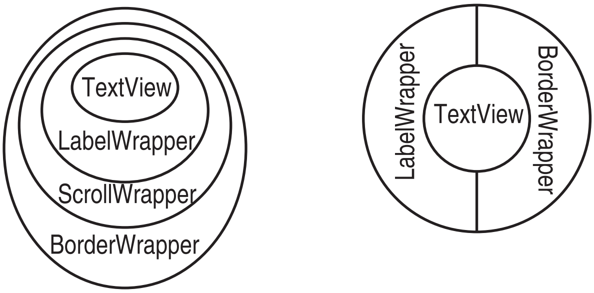
In certain cases, a wrapper may want to conceal part of the wrappee from clients. For example, a ConfidentialWrapper and its wrappee should not be serializable for confidentiality reasons. Thus, the wrapper wants to conceal the interface Serializable from clients in case the wrappee implements it. For this case, a conceals clause may be useful in combination with wraps:

class ConfidentialWrapper **wraps** IView **conceals** Serializable {...}

With this definition, no instance of a ConfidentialWrapper aggregate will ever be an element of Serializable.

Alternatively, a wrapper could be transparent for explicitly listed types only:

class SpecialWrapper **wraps** IView **hoists** IText, IGraphics {...}



a) Conjunctive wrapping b) Disjunctive wrapping

Fig. 7. Conjunctive and Disjunctive Wrapping

When such a SpecialWrapper wraps an instance of a class implementing IText then the functionality declared in IText can be accessed through the wrapper. On the other hand, if the same class also implements another interface, say IContainer, the latter's functionality cannot be accessed through the wrapper and the wrapper cannot be assigned to a variable of static type IContainer. Although transparency is restricted, this approach differs from the containment approach (Sect. 3) in that the type of the aggregate depends on the actual type of the wrappee.

Concealment may be practical for special cases, but it causes type soundness problems because the aggregate is not a subtype of the wrappee. Existing references to the wrappee cannot be redirected to the wrapper, if the latter conceals (part of) the static type of the variable containing the reference. Concealment also causes similar problems in combination with solutions 2 and 3 of applying a wrapper to an already wrapped object (Sect. 5.6). Furthermore, with delegation self calls of the wrappee to methods that are concealed by the wrapper fail. For this, a workaround would be to conceal types only from clients, but not from the aggregate itself.

These problems may, but do not necessarily occur in a given system that uses concealment. In analogy to Eiffel allowing subclasses to conceal[4] inherited members, we could allow concealment of types. This would, however, require system validity checks of complete systems.

5.8 Multiple Wrappees

So far, we have assumed that a given wrapper instance wraps exactly one object. This could be generalized to a fixed or arbitrary number of objects, thereby providing a single view of a subsystem implemented by multiple objects corresponding to the facade pattern [9]. Similar to multiple code inheritance, this works well unless different wrappees implement methods with the same signature and the wrapper does not override them. In this case, message lookup needs to be redefined. With run-time wrapping, such conflicts caused by type transparency may not be visible at compile time.

6 Interaction with Other Typing Mechanisms

In this section we discuss the interaction of generic wrappers with other common typing mechanisms.

6.1 Subclassing

Here we investigate whether and how generic wrappers can substitute inheritance and how the two may be combined.

[4] This is called 'hiding' in Eiffel. We don't use this term here to avoid confusion with Java style hiding of class methods and fields (Sect. 5.2).

Generic wrappers as a substitute for inheritance. If we choose delegation (Sect. 5.3) for generic wrappers, then they can be used to simulate class-based inheritance as follows:

```
class D extends C {...};
D d=new D();
```

Inheritance

```
class D wraps C {...};
D d=new D<new C()>();
```

Simulation with generic wrappers

The main difference is that at compile time we only know the lower bound of the wrappee type for generic wrappers, whereas with inheritance we know the exact superclass. This can be interpreted as flexibility or as lack of knowledge.

If, on the other hand, we use forwarding instead of delegation for generic wrappers, then we cannot modify the semantics of self calls in methods of the supertype. Thus, such generic wrappers cannot be used to simulate inheritance.

Subclassing of wrapper classes. In most mainstream languages subclassing implies subtyping. For this principle to extend to wrappers, a subclass of a wrapper class has to be declared to wrap the same type as its superclass or a subtype of its superclass' static wrappee type. Covariant specialization of the static wrappee type is possible unless the wrappee can be replaced using a method like setWrappee(StaticWrappeeType w), where the static wrappee type occurs in a contravariant position.

6.2 Method Header Specialization and Final Classes

Some languages allow overriding methods to have more specialized headers. For example, Java allows a non-final method to be overridden by a final one and allows the overriding method to have a more restricted exception throws clause and a higher accessibility. Other languages also allow covariant return type and contravariant parameter type specialization.

This creates problems with overriding by wrappers, even if the overriding is statically visible. Wrapping an instance of B with a BWrapper in Fig. 8, would override the final method B.p with an empty throws clause by BWrapper.p, which may throw SomeException.

To prevent such unsound overriding, we have to use wrap-time exceptions. Method header specialization is the type correspondent of semantic refinement discussed in Sect. 5.1.

Final classes pose a similar problem. They should not be subtyped. Thus, it is a compile-time error to declare a wrapper with a static wrappee type that is a final class type. At run time, an exception is thrown if an attempt is made to wrap an instance of a final class.

6.3 Parametric Types

Generic wrappers and parametric types can be combined without problems. Instances of generically derived classes don't distinguish themselves from instances of normal classes. Hence, they can be normally wrapped. Generic wrappers can also be used as bounds in generic classes and as actual parameters in generic derivations.

```
interface IB {
  void p() throws SomeException;
}

class BWrapper implements IB wraps IB {
  public void p() throws SomeException {...};
}

class B implements IB {
  public final void p() {...};
}

IB b=new B();
BWrapper w=new BWrapper<b>(); // illegal wrapping caught by exception
((B)w).p() // final method would be overridden and exception might be thrown
```

Fig. 8. Method Header Specialization Example

7 Generic Wrappers in Java

As a proof of concept, we add generic wrapping to Java. We present generic wrappers as a strict extension, that is existing Java programs need not be changed and instances of existing classes can be wrapped.

We select a consistent set of features from the aforementioned design choices and give a definition of generic wrappers in Java. We base our choices on the motivating examples and the above discussions, without repeating them. Next, we discuss selected integration issues with the Java library. Finally, we show how the defined mechanism solves the motivating problem. A discussion of efficient implementation strategies is beyond the scope and page limit of this paper.

7.1 Feature Selection and Language Integration

Both compile-time and wrap-time overriding and hiding are governed by the same rules as (compile-time) overriding and hiding in subclasses. Furthermore, we don't allow instances of final classes to be wrapped. Violations of these rules by the wrapper/static wrappee pair are flagged at compile time; violations by the wrapper/actual wrappee pair cause exceptions at the time of wrapping.

To get loose coupling between the wrapper and the wrappee and to facilitate semantic reasoning we chose forwarding over delegation and fix the wrappee for the lifespan of the wrapper. All existing references to the wrappee are redirected to the wrapper upon wrapping. We define this in instance method of the wrappee to refer to the (outermost) wrapper except when used for member access.

In a tribute to flexibility, we allow clients to explicitly attain direct references to the wrappee. The implementor of the wrapper class determines whether clients can get direct references to the wrappee by putting an access modifier (private, protected, public) between the keyword wraps and the static wrappee type, e.g:

```
class LabelWrapper3 wraps public IView {...}
```

The access modifier of the wrappee in a subclass must provide at least as much access as that in the superclass. The keyword wrappee can be treated like the name of a final instance field of the wrapper class with the used modifier, e.g public in the above example. To navigate back from a wrappee to its outermost wrapper, the method:

```
public final Object getWrapper() {return this;}
```

is added to the class Object. With the above definitions, this method returns a reference to the wrapper if the receiver object is wrapped and otherwise to the receiver itself.

We allow only conjunctive, but not disjunctive wrapping. Wrapping an already wrapped object corresponds to wrapping its outermost wrapper. Because it is not sound in combination with the above features, we don't allow concealment. Every wrapper has exactly one wrappee.

7.2 Library Integration

The library being an integral part of Java —the description of three packages is even part of the Java language specification— we discuss how serialization and cloning interplay with generic wrappers. Generic wrappers integrate in a straightforward way with most libraries, often providing new possibilities.

For instances of a Java class to be serializable, the class must implement the empty interface Serializable. A problem arises if only the wrapper or only the wrappee implement Serializable. In this case, the aggregate appears to be serializable although it isn't. In practice, the best solution is to throw a NotSerializableException when trying to serialize such an aggregate. In [4], we discuss certain options to statically avoid part of the problem.

Similar problems occur with cloning if only the wrapper or only the wrappee implements Cloneable. Because we don't allow disjunctive wrapping, clone has to create deep copies. Throwing a CloneNotSupportedException is again the best solution.

7.3 Assessment

Our mechanism fulfills all requirements (Fig. 2) except for genericity (2). The latter fails in cases where overriding would not be sound. We consider this acceptable because exceptions are already thrown at the time of wrapping —and not at the time of member access— and because creation of new instances can also fail for other reasons with an exception in existing Java.

Clearly, the motivating problems (Sect. 2.1) can be solved with the presented generic wrappers for Java.

8 Type Soundness

In this section, we report on a mechanically verified formal proof of type soundness of Java extended with generic wrappers. Type soundness intuitively means that all values produced during any program execution respect their static types. An immediate corollary of type soundness is that method calls always execute a suitable method, that is, there are no 'method not understood' errors at run time.

Our proof of type soundness for generic wrappers is based on the work of von Oheimb and Nipkow [32]. They have formalized a large subset of Java and mechanically proved type soundness with the theorem prover Isabelle/HOL [23].

For this paper, we have added generic wrappers to this formalization,[5] adapted the proofs, and ran them through Isabelle/HOL. Here, we present the widening relations applicable to generic wrappers. A full report of all the mechanical details is beyond the scope of this paper.

The Java language specification [10] introduces identity and irreflexive widening conversions separately. 'Widening' is Java's form of subtyping. Since identity conversions are possible in all conversion contexts permitting widening, the two are merged in the formalization. The expression $\Gamma \vdash S \preceq T$ says that in program environment Γ objects of type S can be transformed to type T by identity or widening conversion. In particular, expressions of type S can be assigned to variables of type T and expressions of type S can be passed as formal parameters of type T.

We use the following naming conventions:

C, D	classes	A	list of classes
I, J	interfaces	S, T	arbitrary types
R	reference type	Γ	program, environment

The judgment $\Gamma \vdash C \prec_{\mathsf{C}} D$ expresses that class C is a subclass of class D, $\Gamma \vdash C \rightsquigarrow I$ that class C implements interface I, and $\Gamma \vdash I \prec_{\mathsf{I}} J$ that I is a subinterface of J. Furthermore, $\mathsf{is\_type}\ \Gamma\ T$ expresses that T is a legal type in Γ, $\mathsf{RefT}\ R$ denotes reference type R, and NT stands for the null type.

$\mathsf{Class}\ C$ stands for the class type C and $\mathsf{Iface}\ I$ for the interface type I. In our formalization we now have two kinds of classes: normal (non-wrapper) classes and wrapper classes. The discriminator $\mathsf{is\_wrapper}\ \Gamma\ C$ is true if C is a wrapper class. $\mathsf{WrappeeOf}\ \Gamma\ C$ denotes the the static wrappee type of class C in program Γ.

At run time, instances of wrapper classes are of aggregate types. Aggregate types are finite lists of at least two class types. An instance of the wrapper class C wrapping an instance of the wrapper class D that itself wraps an instance of the (non-wrapper) class E belongs to type $\mathsf{Aggregate}\ [C, D, E]$. The discriminator $\mathsf{is\_aggregate}\ \Gamma\ A$ is true if A denotes a possible combination of classes for an aggregate. Since there are no variables of aggregate type and because we do not allow the dynamic reassignment of wrappees, we only need widening rules with aggregates on the left-hand side of the conclusion judgment.

Furthermore, the discriminators $\mathsf{is\_class}\ \Gamma\ C$ and $\mathsf{is\_iface}\ \Gamma\ I$ are used. The following six typing judgments apply unchanged also to wrapper classes:

$$\frac{\mathsf{is\_type}\ \Gamma\ T}{\Gamma \vdash T \preceq T} \qquad \frac{\mathsf{is\_type}\ \Gamma\ (\mathsf{RefT}\ R)}{\Gamma \vdash \mathsf{NT} \preceq \mathsf{RefT}\ R}$$

[5] At http://www.abo.fi/~mbuechi/publications/GenericWrappers.html the Isabelle theories are available.

$$\frac{\Gamma \vdash I \prec_{\mathsf{i}} J}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Iface}\ J} \qquad \frac{\mathsf{is\_iface}\ \Gamma\ I; \mathsf{is\_class}\ \Gamma\ \mathsf{Object}}{\Gamma \vdash \mathsf{Iface}\ I \preceq \mathsf{Class\ Object}}$$

$$\frac{\Gamma \vdash C \prec_{\mathsf{C}} D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D} \qquad \frac{\Gamma \vdash C \leadsto J}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ J}$$

The following widening rules involving wrapper classes are used at compile time:

$$\frac{\mathsf{is\_wrapper}\ \Gamma\ C;\ \Gamma \vdash \mathsf{WrappeeOf}\ \Gamma\ C \preceq \mathsf{Class}\ D}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_wrapper}\ \Gamma\ C;\ \Gamma \vdash \mathsf{WrappeeOf}\ \Gamma\ C \preceq \mathsf{Iface}\ J}{\Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Iface}\ J}$$

The following widening rules involving aggregates are used at run time (set converts a list into a set):

$$\frac{\mathsf{is\_aggregate}\ \Gamma\ A;\ \exists C \in \mathsf{set}\ A. \Gamma \vdash \mathsf{Class}\ C \preceq \mathsf{Class}\ D}{\Gamma \vdash \mathsf{Aggregate}\ A \preceq \mathsf{Class}\ D}$$

$$\frac{\mathsf{is\_aggregate}\ \Gamma\ A;\ \exists C \in \mathsf{set}\ A. \Gamma \vdash C \leadsto J}{\Gamma \vdash \mathsf{Aggregate}\ A \preceq \mathsf{Iface}\ J}$$

The main advantages of a mechanized over a paper-and-pencil proof are additional confidence and better support for extensions. We would like to stress the second aspect. Not only did the formalization result in a soundness proof, but the proof tool also reminded us of what all needed to be defined about generic wrappers before the desired properties could be established. Most proof scripts worked without modifications. The fact that all theorems were reproved mechanically[6] for the extended language definition conveys more confidence than the typical adaptation of a paper-and-pencil proof with 'this-should-still-hold' handwaving.

9 Related Work

Section 3 already provides an overview of some related mechanisms. With the exception of delegation, where a final comparison with our mechanism is deemed interesting, these technologies are not discussed again here. Comparisons with less closely related language mechanisms as well as binary component standards can be found in [4].

Delegation in prototype-based languages. What do we gain with generic wrappers over delegation in prototype-based languages? First, the static wrappee type and calls to it can be statically type checked. Some prototype-based languages, such as Cecil [5], also have (optional) static type systems. However, these languages require the exact type or even the concrete instance of the parent object to be known at compile time. The same approach is taken by prototype-based object calculi, e.g. [7]. Thus, they fail the requirement of run-time applicability (1).

[6] At the time of writing, a few lemmata have not yet been mechanically proved.

Second, with generic wrappers the dynamic wrappee type can be checked with run-time type tests. Third, type casts are the only points of failure; method lookup always succeeds. This greatly simplifies debugging by indicating errors closer to where they occur. Fourth, generic wrappers are targeted at mainstream class-based languages.

For our exemplary generic wrappers in Java, we have chosen a set of distinguishing features that facilitate modular reasoning. First we have forwarding rather than delegation. Second the wrappee is assigned snappily differentiating it from reassignable parent fields. Third, we disallow disjunctive wrapping. The latter is no problem because we get sharing of behavior from classes whereas prototype-based languages have to use shared parents for this.

Lava. Kniesel [15] has implemented an extension of Java with wrappers. The main difference to our generic wrappers is that in his proposal the aggregate is not a subtype of the actual, but only of the static wrappee type. Thus his proposal fails the transparency requirement (3) and is more limited in its applicability. Lava's wrappers are a form of the decorator pattern with automatically generated forwarding stubs and multiple wrappees combined with delegation. Wrappees can be reassigned, thereby, complicating semantic reasoning. The proposal is not type sound because the wrappees are assigned within the constructor. Independent extensibility, the focus of our proposal, is not well supported.

Delegation for software and subject composition. Harrison et al. [11] discuss options for different bindings of this in the decorator and facade patterns. They show how to implement delegation using either stored or passed pointers in class-based languages. Furthermore, they propose a declarative approach, to be used by component assemblers, permitting the binding of this to be customized on a per-method base. Their solution does not address the shortcomings of the decorator pattern with respect to our requirements. Namely, it does not provide for transparency (3).

Dynamic object specialization and reflective mix-ins. gbeta [6], a generalized version of BETA, supports two forms of dynamic inheritance through multiple inheritance. Dynamic object specialization is a dynamic modification of the structure of an existing object, preserving object identity. For example, the statement somePtn##->anObject## enhances the structure of anObject with the pattern somePtn. Furthermore, gbeta allows non-constant virtual types as superpatterns.

Because gbeta uses submethoding with INNER rather than overriding, it is not obvious how the mechanisms of gbeta could be transferred to more 'standard' object-oriented languages.

Mezini [18] presents a sophisticated, but complex approach to object evolution without name collisions. However, her work is untyped. Steyaert et al. [27] propose dynamic inheritance through mix-ins. The catch is that each object must contain a specification of all its potential enhancements. This renders their proposal inapplicable for mutually unaware component vendors.

A proposal for mix-ins, which allow types to be derived at run time, is presented in [4]. It is shown that —ignoring the use of the type parameter in places other than the extends clause— they correspond to a special kind of generic wrappers where the wrappee must be created along with the wrapper.

Objective C. Categories in Objective C [20] allow classes to be extended with a new set of methods/protocols independently of the original class definition. This compile-time mechanism corresponds to creating a subclass and globally replacing all occurrences of the superclass by the subclass. Categories modify whole classes, rather than individual objects. Categories do not fulfill the requirements of run-time applicability (1) and genericity (2).

Binary Component Adaption [13] provides for similar adaption of Java binaries as categories for Objective-C binaries. With respect to the problem at hand, it has the same shortcomings.

Aspect-oriented programming. Aspects [14] are a new category of programming construct that 'cross-cut' the modularity of traditional programming constructs. So an aspect can localize, in one place, code that deeply affects the implementation of multiple classes or methods. Aspects modify classes at compile time. Hence, they do not address the problems of run-time composition of objects created by different components from different vendors.

Mix-in calculus. Bono et al. have developed a formal calculus of classes and mix-ins [2]. Method declarations in mix-ins are explicitly marked as overriding an existing method or introducing a new method. The lower type bound (static wrappee type) is computed from the signature of a mix-in. Redefined methods give positive type information and new methods negative type information. Subtyping is determined by the types' structures. Negative type information is used to avoid mix-in-application-time exceptions.

10 Conclusions

Late composition of software components from different vendors is the essence of component software, enabling component markets and flexible reuse. One form of late composition is the combination of features implemented by different vendors into object-aggregates that appear as single objects to their clients. Our analysis shows, that existing technologies fail to fully unlock this power.

To remedy the problem, we have proposed generic wrappers, a typed form of dynamic inheritance. We have analyzed the design space with respect to both type soundness and semantic intuition, desirability, and consistency with existing mechanisms, such as subclassing. One options is forwarding instead of delegation to loosen the coupling and, thereby, avoid the semantic fragile base class problem. Another options is the snappy assignment of the wrappee to facilitate modular semantic reasoning.

As a proof of concept, we have chosen a consistent set of desirable features for a concrete mechanism, which we added to Java. Finally, we have given a mechanized proof of type soundness for the extended language. Additionally, the formalization provides an operational semantics for Java extended with generic wrappers.

Acknowledgments. David von Oheimb and Tobias Nipkow provided us with their formalization of Java and helped us with our extensions. We would like to thank Ralph Back, Dominik Gruntz, and Cuno Pfister for a number of fruitful discussions. The referees' helpful comments are also gratefully acknowledged.

References

1. Pierre America. Designing an object-oriented programming language with behavioral subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop*, pages 60–90. LNCS 489, Springer Verlag, 1991.
2. Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proceedings of ECOOP '99*, pages 43–66. LNCS 1628, Springer Verlag, 1999.
3. Martin Büchi and Wolfgang Weck. Compound types for Java. In *Proceedings of OOPSLA '98*, pages 362–373. ACM Press, 1998. http://www.abo.fi/˜mbuechi/.
4. Martin Büchi and Wolfgang Weck. Generic wrapping. Technical Report 317, Turku Centre for Computer Science, March 2000. http://www.abo.fi/˜mbuechi/.
5. Craig Chambers. The Cecil language: Specification & rationale (version 2.1). Technical report, University of Washington, March 1997.
6. Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.
7. Kathleen Fisher and John C. Mitchell. Notes on typed object-oriented programming. In *Proceeding of Theoretical Aspects of Computer Software*, pages 844–885. LNCS 789, Springer Verlag, 1994.
8. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 171–183. ACM Press, 1998.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, 1996.
11. William Harrison, Harold Ossher, and Peri Tarr. Using delegation for software and subject composition. Technical Report RC-20946 (92722), IBM Research Division, T.J. Watson Research Center, August 1997.
12. John Hogg. Islands: Aliasing protection in object-oriented languages. In *Proceedings of OOPSLA '91*, pages 271–285. ACM Press, 1991.
13. Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proceedings of ECOOP '98*, pages 307–329. LNCS 1445, Springer Verlag, 1998.
14. Gregor Kiczales et al. Aspect-oriented programming. In *Proceedings of ECOOP '97*, pages 220–242. LNCS 1241, Springer Verlag, 1997.
15. Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of ECOOP '99*. LNCS 1628, Springer Verlag, 1999.
16. Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA '86*, pages 214–223. ACM Press, 1986.
17. Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, second edition, 1992.
18. Mira Mezini. Dynamic object evolution without name collisions. In *Proceedings of ECOOP '97*, pages 190–219. LNCS 1241, Springer Verlag, 1997.
19. Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of ECOOP '98*, pages 355–374. LNCS 1445, Springer Verlag, 1998.
20. NeXT Software, Inc. *Object-Oriented Programming and the Objective-C Language*. Addison-Wesley, 1993.
21. Object Management Group. CORBA components, 1999. Revision February 15, 1999, formal document orbos/99-02-01, http://www.omg.org.
22. Geoff Outhred and John Potter. Extending COM's aggregation model. In *Component-Oriented Software Engineering Workshop (in conjunction with the Australian Software Engineering Conference)*, 1998.

23. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. LNCS 828, Springer Verlag, 1994.
24. Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. 16th ACM Symp. Principles of Programming Languages*, pages 242–249. ACM Press, 1989.
25. Dale Rogerson. *Inside COM*. Microsoft Press, 1996.
26. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA '86*, pages 38–45. ACM Press, 1986.
27. Patrick Steyaert and Wolfgang De Meuter. A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP '95*, pages 127–144. LNCS 952, Springer Verlag, 1995.
28. Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, third edition, 1997.
29. Sun Microsystems, Inc. Java Beans, 1997. http://java.sun.com/beans/.
30. Clemens A. Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
31. D. Ungar and R.B. Smith. Self: The power of simplicity. In *Proceedings of OOPSLA '87*, pages 227–241. ACM Press, 1987.
32. David von Oheimb and Tobias Nipkow. Machine-checking the Java specification: Proving type-safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, pages 119–156. LNCS 1523, Springer Verlag, 1999.

Copying and Comparing: Problems and Solutions

Peter Grogono[1] and Markku Sakkinen[2]*

[1] Department of Computer Science, Concordia University
Montreal, Quebec
grogono@cs.concordia.ca

[2] Software Systems Laboratory, Tampere University of Technology
Tampere, Finland
sakkinenm@acm.org

Abstract. In object oriented programming, it is sometimes necessary to copy objects and to compare them for equality or inequality. We discuss some of the issues involved in copying and comparing objects and we address the problem of generating appropriate copying and comparing operations automatically, a service that is not provided by most object oriented languages and environments. Automatic generation appears to be not only desirable, because hand-coding these methods is mechanical and yet error-prone, but also feasible, because the form of the code is simple and largely predictable.

Some languages and some object models presented in the literature do support generic copying and comparing, typically defining separate "shallow" and "deep" versions of both operations. A close examination of these definitions reveals inadequacies. If the objects involved are simple, copying and comparing them is straightforward. However, there are at least three areas in which insufficient attention has been given to copying and comparing complex objects: (1) values are not distinguished from objects; (2) aggregation is not distinguished from association; and (3) the correct handling of linked structures other than trees is neglected.

Solving the third problem requires a mechanism built into the language, such as exists in Eiffel. Building such a mechanism without modifying the language requires a language with sufficient reflexive facilities, such as Smalltalk. Even then, the task is difficult and the result is likely to be insecure.

We show that fully automatic generation of copying and comparing operations is not feasible because compilers and other software tools have access only to the structure of the objects and not to their semantics. Nevertheless, it is possible to provide default methods that do most of the work correctly and can be fine-tuned with a small additional amount of hand-coding.

We include an example that illustrates the application of our proposals to C++. It is based on additional declarations handled by a preprocessor.

Keywords: Copying, cloning, equality, complex object structures.

* On leave from the Department of Computer Science and Information Systems, University of Jyväskylä Jyväskylä, Finland

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 226–250, 2000.

© Springer-Verlag Berlin Heidelberg 2000

1 Introduction

In object oriented programming, it is sometimes necessary to copy the contents of one object to another object, make a new copy (or clone) of an object, or to decide whether two objects are equal. Previously [5], one of us (PG) has argued that copying should be needed only rarely in object oriented programs; nevertheless, copying is sometimes necessary, and most object oriented programming languages provide some kind of facilities for copying. Equality testing is certainly sometimes necessary, too, but the details of how to perform the comparison can be subtle.

For the purposes of this discussion, we assume that programmers need to make copies of objects and to compare objects. The two operations are related in that we expect "`X` is a copy of `Y`" to imply "`X` is equal to `Y`", with appropriate interpretations of "copy" and "equal". Complexity arises because the desired meanings of "copy" and "equal" depend on features of the objects involved and on the circumstances in which the operations are used. We will argue:

- there are several plausible meanings that can be assigned to "copy" and "equal";
- a programming language cannot provide appropriate default versions of "copy" and "equal" for all situations; and therefore
- a programming language should provide a set of standard operations that help programmers to construct appropriate copying and comparison operations with minimal effort and maximal security.

Section 2 provides a basis for the discussion. Section 3 discusses the problems involved in copying and comparing objects, including the special problems that arise with cyclic structures. Section 4 describes the facilities for copying and comparing provided by various programming languages. Section 5 demonstrates a possible implementation of our proposals and, finally, Section 6 presents our conclusions.

2 Background

In this section, we introduce an object model that provides a basis for the subsequent discussion and some terminology. The object model can be seen as the "greatest common divisor" of most object oriented programming languages, as simple as possible for presenting the topics of this paper. It is most suited to typed, class-based languages.

In our object model, an object contains zero or more attributes. Each attribute is one of:

- a basic value;
- an object; or
- a reference[1] to an object.

[1] This covers both "reference" and "pointer" in `C++` terminology.

A *basic value* is an indivisible attribute in the object model. In most programming languages, Booleans, characters, and integers are basic values. A string might be considered either as a basic value or as a vector of characters.

The model permits nested objects: an object may be an attribute of another object. Some programming languages (e.g., C++ and Eiffel) provide nested objects but most object-oriented languages allow an object to contain only references to other objects. The model does *not* allow for references to nested objects, a situation that can arise in C++ but not in most other languages. The model permits an object to contain a reference to another object. Cycles are possible: a chain of one of more references may lead back to the original object.

In this paper, we assume that the objects under discussion belong to a single identifier space. The additional complications of copying and comparing in distributed object systems are beyond the scope of the present work.

2.1 Essential and Accidental Attributes

We distinguish essential and accidental attributes of an object.[2] An *essential attribute* is indisputably a part of the object; an *accidental attribute* is another object that is related in some way to the object in question but is not a part of it. For example, if the object in question is an instance of class `Car`, we would consider the attribute `engine` to be essential but the basic value `distanceTravelled` and the reference `owner` to be accidental. The distinction between "accidental" and "essential" is orthogonal to that between "reference" and "containment". The model permits all four possibilities. When an attribute is represented by a reference, it is the referent object itself, not the reference, that is the accidental or essential attribute.

Accidental attributes are intended as a generalization of associations. An association is a "structural relationship between peers" where "peers" are classes at the same conceptual level [3]. An association is a kind of accidental attribute but it is not the only kind. Associations are usually implemented as references to other full-fledged objects, although more elaborate implementations have been proposed. But objects may also contain counters, flags, descriptors, and other attributes that are needed by the application software but are conceptually not part of the object.

The distinction between essential and accidental is not always obvious. As a rule of thumb, the relationship between two objects is an association (and therefore accidental) if destroying one object does not logically entail destroying the other, otherwise one object is an attribute of the other. Similarly, an attribute is accidental if removing it from the object does not destroy the basic integrity of the object.

Example 1. Figure 1 provides a simple if rather contrived example of this terminology. The class declaration introduces a `Detector` which is responsible for monitoring the performance of a pump. Each detector has: a pointer to its own

[2] This distinction is based loosely on Aristotle's categories.

```
class Detector
{
public:
    . . . .
private:
    Counter *counter;
    Clock *clock;
    Pump *pump;
    long startTime;
};
```

Fig. 1. The class `Detector`

`Counter` object, used to count events; and a pointer to a unique `Clock` object, shared by all detectors. It has a reference to the pump it is monitoring and, finally, a basic value `startTime`. The attributes `counter` and `startTime` are essential; the attributes `clock` and `pump` are accidental because they are not part of the object.

2.2 Values and Objects

Following MacLennan [14], we distinguish *values*, which are immutable abstractions, and *objects*, which are containers with mutable attributes. For example, `true` and `false` are immutable, Boolean values. In contrast, a `Switch` object might have a mutable attribute with a Boolean value that, at different times, is either `true` or `false`.

In many languages, there is an implicit assumption that basic values are "values" and that classes define "objects" in MacLennan's sense. This leads to confusion for entities such as `String` which should arguably be implemented as immutable (in accordance with the view that strings are basic values) but is usually implemented as a mutable class for efficiency. For example, Java addresses this confusion by providing both an immutable class, `String`, and a mutable class, `StringBuffer` [2, page 172] and CLU has both mutable and immutable versions of the structured type constructors [12].

Our object model distinguishes *mutable* and *immutable* (`const` in C++). This distinction is orthogonal to that between simple and structured objects. We allow both mutable, simple objects, such as a `Counter` that contains an updatable integer, and immutable structured objects, such as a binary tree with an integer at each node.[3] Figure 2 shows examples of each of the four categories.

References are not values in MacLennan's sense because the meaning of a reference depends on the existence of its referent. Many formal object models postulate a fixed, given set of object identifiers, but they must then prohibit the use of meaningless identifiers by suitable integrity constraints. By contrast,

[3] Of course, binary trees can also be represented with mutable objects.

	Mutable	**Immutable**
Simple	`Switch`	`Integer`
Structured	`Person`	`BinaryTree`

Fig. 2. An orthogonal classification

one would not imagine a constraint saying that, depending on the total state of the object system, some integer values must not appear as any attribute of any object.

An immutable object may in general contain references to mutable objects. We use the term *strictly immutable*[4] for the important special case in which this is not allowed. The contents of a strictly immutable object is a *pure value.*

Example 2. The distinction between mutable and immutable is useful but has some subtle ramifications. Consider an application with an immutable class `Rectangle` with integer attributes `length` and `width`. An application can save space by creating one instance of each desired size of rectangle and sharing these instances amongst clients.

Suppose that the same application has a class `Point` with mutable integer attributes `x` and `y`. An instance of class `GraphicalObject` is an object with attributes `shape`, referencing a `Rectangle`, and `position`, referencing a `Point`. Using these classes, the application can create structures like that shown in Figure 3, in which `GraphicalObjects` share `Rectangles` and `Points`. The instances of `Rectangle` are immutable objects; that they are shared is undetectable to the program. The instances of `Point` are mutable objects; that they are shared is crucial to the behaviour of the system, because changing the coordinates of a `Point` will cause movement of all of the `GraphicalObjects` referencing it.

The use of two or more names to refer to a single object is called "aliasing" and is sometimes considered to be undesirable. Sharing of both mutable and immutable objects is an important feature of object modelling [5]. Certainly, unintended aliasing can cause serious and subtle errors. But, in many situations, multiple references to a single object are the most appropriate way to model real-world relationships.

2.3 Abstract and Concrete Values

For many objects, it is useful to distinguish the *abstract value* of the object from its *concrete representation.* For example, if the object is a set, its abstract value consists of the members of the set, but its concrete representation might be an array, a list, a hash table, or some other kind of data structure. In CLU, this idea is elevated to a principle [13], and the language provides explicit syntax

[4] "Deep-immutable" would also be suitable.

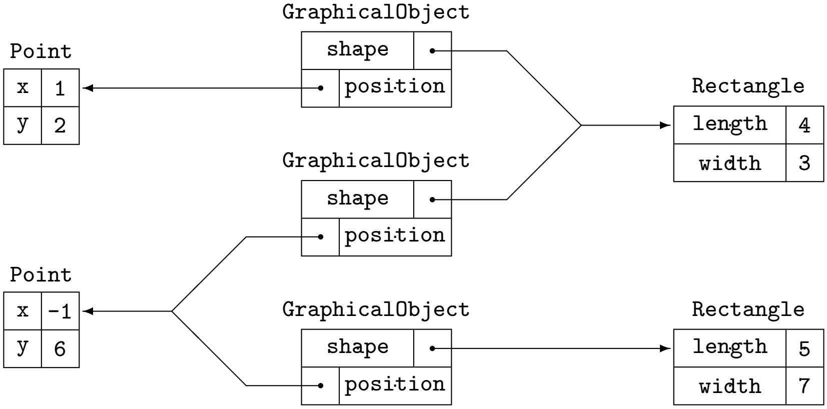


Fig. 3. Using `GraphicalObjects`

for switching between the interface, or external view, and the representation, or internal view. (We avoid the word "conversion" because abstract values do not exist directly either in the language or in programs.) Most object oriented programming languages, however, do not provide explicit syntax for this distinction, and the distinction between abstract values and their representations is mostly in the mind of the programmer.

It is quite natural to think of abstract values in connection with comparison operations. Suppose, for example, that we have two objects `X` and `Y`, each representing sets, and each using an unordered list to represent a set. We would consider `X` and `Y` to be equal if their lists were, respectively, $[2, 1, 3]$ and $[3, 1, 2]$, because both lists share the abstract value $\{1, 2, 3\}$.

It is perhaps less obvious, but equally important, that copying operations should respect abstract values. For example, we might want to represent a set as a binary search tree while we are building it and as an ordered array for retrieval operations. The conversion from tree to array could be accomplished by a copy operation that recognized the need for a change of representation but preserved the abstract value of the set.

2.4 Inheritance

It is usually best not to inherit copying and comparison operations from superclasses because the inherited methods do not process attributes introduced by the subclass. To use a familiar example, if the subclass `ColouredPoint` inherits its copy and comparison methods from the superclass `Point`, the attribute `colour` will be neither copied nor compared. Nevertheless, a copy or compare method in the subclass can often make use of the corresponding method in the superclass and should do so if the programming language provides an appropriate mechanism, as most object oriented programming languages do.

If the programming language supports multiple inheritance, copy and comparison methods from one or more superclasses can be used to construct the corresponding methods in the subclass. Object oriented programming languages with multiple inheritance typically provide appropriate mechanisms for calling methods from multiple superclasses. A few languages provide standard method combinations that do not require explicit invocation: for example, "before methods" and "after methods" in CLOS [7, page 50].

With either single and multiple inheritance, the programming language can provide some support for implementing copy and comparison methods in subclasses. Our approach, described in Section 5.3, avoids some of the common problems, such as covariant redefinition of equality. It is clear, however, that the programming language cannot generate the subclass methods correctly in all cases.

3 Copying and Comparing

In this section, we discuss copying and comparing operations in detail, the consequences of cycles in structures, and the problems of operations between instances of different classes.

3.1 Copying

The word "copy" is used loosely to mean several different things. In this paper, we use particular words for various copying operations, as follows.

- *Assign* means "update a reference" and does not involve copying the contents of objects.
- *Replace* means "copy data from an object into another object that already exists".
- *Clone* means "create a new object and copy data from an existing object into it".

The object from which values are obtained is called the *source object*. The object that is changed or created by the copy operation is called the *target object*.

To clarify these definitions, suppose we have the situation shown in the left part of Figure 4 The letters `X` and `Y` are variable names in a program text; the boxes are run-time objects; the letters $\mathcal{A}$ and $\mathcal{B}$ indicate the values of the objects' attributes; and the arrows indicate the relation between names and objects. The target object is `X` and the source object is `Y`. The right part of Figure 4 shows the various situations that can arise after different kinds of copying operations. $\mathcal{B}'$ indicates a fresh copy of the value $\mathcal{B}$. We have used a neutral, procedural syntax for each operation. In each case, the first argument is the name of the target object. We note that:

- the operation `assign` (reference assignment) creates an alias — afterwards, `X` and `Y` both refer to the same object;

– after the operations `assign` and `clone`, the reference `X` has changed and the object $\mathcal{A}$ becomes inaccessible, or "garbage", unless there are other references to it; and
– `clone` could be implemented by first creating a new, uninitialized object and then using `replace` to initialize its attributes.

In `assign`, the type of Y can be a subtype (subclass) of the type of X (inclusion polymorphism). For `replace`, we require the objects `X` and `Y` to be of the same type; this restriction will be discussed in Section 3.4.

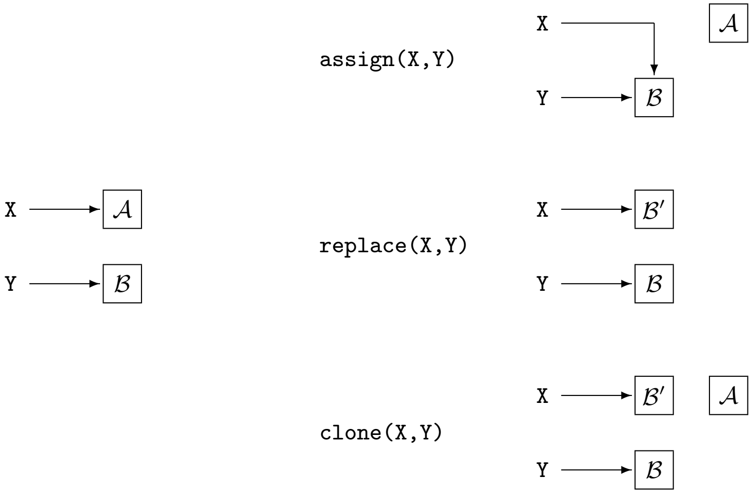


Fig. 4. Copying Operations: before (left) and after (right)

We can further categorize copying operations by their "depth". A *shallow* operation copies, but does not trace, references. A *deep* operation traces references and applies a copy operation to their referents. The distinction between shallow and deep does not apply to reference assignment. A "shallow replace" operation replaces attributes in the target object but not in objects referenced by the target object. A "deep replace" operation replaces non-reference attributes in both the target object and in objects referenced by the target object. Deep replacement requires the source and target object to be isomorphic structures. For example, deep replacement of one list by another list would require both lists to have the same number of items. It would be possible to provide a deep replacement operation that succeeded if the source and target had isomorphic structures and failed otherwise.

Swapping can be seen as a generalized form of replacing. The naive implementation (which requires a temporary intermediate object, one clone operation, two copy operations, and a deletion) is inefficient for complex objects. A more efficient, customized swapping method could be implemented automatically.

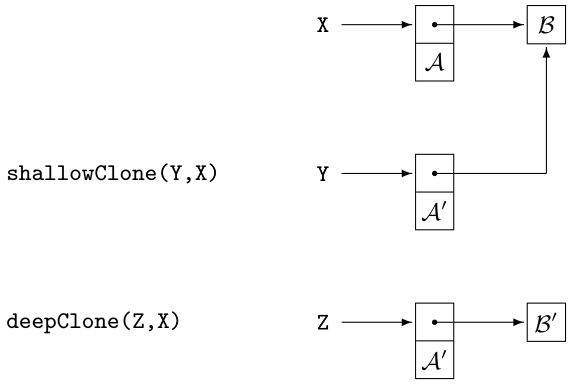


Fig. 5. Shallow and deep clones

Shallow cloning and deep cloning are distinct operations. In Figure 5, `Y` is a shallow clone of `X`. As in Figure 4, primes indicate newly created values. Note that the object $\mathcal{B}$ is shared by `X` and `Y`: shallow cloning may introduce aliasing. `Z` is a deep clone of `X`; an object and its deep clones should normally be disjoint. Approximately the foregoing definitions of shallow and deep cloning have been used in the literature of object oriented systems for many years [8]. Unfortunately, the definition of deep cloning breaks down when cyclic structures are involved! We discuss the application of these definitions to cyclic structures in Section 3.3.

In principle, there is an infinite number of possible ways of cloning a structure. We can define $\mathtt{clone}^K(\mathtt{X}, \mathtt{Y})$ as follows: $\mathtt{clone}^0(\mathtt{X}, \mathtt{Y})$ is the same as $\mathtt{assign}(\mathtt{X}, \mathtt{Y})$; and $\mathtt{clone}^K(\mathtt{X}, \mathtt{Y})$ for $K > 0$ creates a new object and assigns it to `X`, copies value attributes, and performs $\mathtt{clone}^{K-1}$ on reference attributes.

Languages that provide cloning operations usually provide $\mathtt{clone}^1$ (shallow copy) or $\mathtt{clone}^\infty$ (deep copy). However, the Smalltalk method `deepCopy`, described in Section 4.6, performs $\mathtt{clone}^2$. The shallow and deep operations are not generally useful. In most cases, "shallow" is too shallow[5] and "deep" is too deep. In order to be generally applicable, copying operations should respect the semantic properties of objects rather than merely their syntactic properties.

3.2 Comparing

There are two principal ways of comparing objects: we can check equality or identity. If `X` and `Y` are objects that have similar structure and whose corresponding attributes are equal, we say that `X` is equal to `Y`. Note that this definition is recursive but terminating: we are defining equality of complex objects in terms of equality of simpler objects. An identity comparison, on the other hand, de-

[5] Embedded objects can be at least a partial help.

termines whether the names X and Y refer to the same object; the attributes of X and Y are not even considered.

In their description of CLU, Liskov and Guttag state that “it should be impossible to distinguish between equal objects” [13, page 93]. For mutable objects, this implies that equality is identity. Otherwise, it would be possible to distinguish apparently equal objects by making a change to one object and then checking whether the other object changed. Two immutable objects are equal if their abstract values are the same. If X and Y refer to immutable objects with equal attributes, the programmer cannot determine whether X and Y refer to the same object or to different objects with the same value.

As with copying, we can define shallow, deep, and “depth-K” equality comparisons in the sense of the preceding section. A shallow comparison compares references but does not trace them, a deep comparison traces references and recursively compares their referents, and a “depth-K” comparison traces at most K pointers from the original object. As with copying, this conventional but naive approach works only as long as the object structures do not contain cycles.

We use four binary operators to denote comparison relations:

$\mathtt{X} =^0 \mathtt{Y}$ means “X and Y are references to the same object”;
$\mathtt{X} =^1 \mathtt{Y}$ means “X and Y are shallow equal”;
$\mathtt{X} =^\infty \mathtt{Y}$ means “X and Y are deep equal”;
$\mathtt{X} \cong \mathtt{Y}$ means “X and Y are structurally equal”.

Figure 6 compares the evaluation of these relations for different pairs of objects. We note that identity implies shallow equality and shallow equality implies deep equality. In general, equality at depth K implies equality at all depths greater than K. Examples (4) and (5) show that deep equality and structural equality are independent. The distinction between deep and structural equality has rarely been mentioned in the literature. Even such recent work as [1] does not consider structure equality at all but just studies three different formalizations of conventional deep equality.

3.3 Cyclic Structures

Objects that contain direct or indirect references to themselves introduce a further complication into copying and comparing. Figure 7 is a simplified version of a diagram by Meyer [17, page 249]. The original object is X; the objects Y and Z represent possible outcomes of making a copy of X. Using our definitions:

- $\mathtt{Y} =^1 \mathtt{X}$, implying $\mathtt{Y} =^\infty \mathtt{X}$, but not $\mathtt{Y} \cong \mathtt{X}$;
- Y is a shallow clone of X; and
- $\mathtt{Z} =^\infty \mathtt{X}$ and $\mathtt{Z} \cong \mathtt{X}$.

According to the definitions of Section 3.1, a deep clone of X would be an infinite list.

	$X =^0 Y$	$X =^1 Y$	$X =^\infty Y$	$X \cong Y$
(1)	true	true	true	true
(2)	false	true	true	true
(3)	false	false	true	true
(4)	false	false	false	true
(5)	false	false	true	false

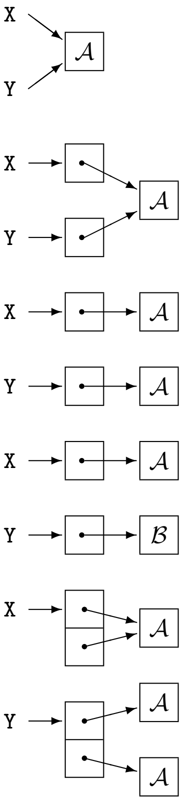


Fig. 6. Four kinds of equality

According to the definitions of Eiffel [16,17], in contrast, Z is a deep clone of X. Clearly, a complete implementation of deep cloning in Eiffel must detect cyclic references and create corresponding cyclic references in the target object.

A similar problem occurs when the object structure contains no (directed) cycles but there are several distinct paths to one object. In this case, conventional deep copying does not get into infinite recursion but produces a target object with a different structure than the source object. For example, in Figure 6(5), Y is a deep copy of X.

The distinction between shallow and deep cloning, at first glance, seems to be related to the preservation of structure but is in fact almost orthogonal. To see this distinction, observe that, in Figure 7:

- Y and X have the same reference attribute values but different structures;
- Z and X have the same structures but different reference attribute values.

One could argue that even the definition of shallow cloning and shallow equality should be modified to take self-references into account. Such structural

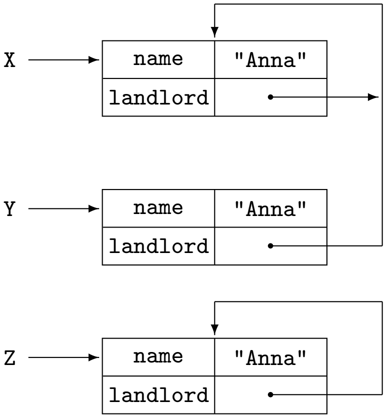


Fig. 7. The problem of self-reference

shallow equality would imply structural equality, just as conventional shallow equality implies deep equality. In that case, Z would be a structural shallow clone of X but Y would not be. Similarly, structural variants of depth-K cloning and equality can be defined.

In general, we can draw a distinction between *value-preserving* cloning operations and *structure-preserving* cloning operations. Which operation is the most appropriate in a particular situation depends on the nature of the application and cannot be deduced simply by examining the objects.

If the kind of deep replacement operations described in the previous section are required, it would be useful to be able to compare the structures of two objects without considering the values of their non-reference components. Because structural comparisons can be expensive, it would also be useful to offer also a three-valued comparison operation: equal by structure and leaf values; equal by structure only; and unequal by structure.

Meyer [16, 305–307] notes that comparison of cyclic structures introduces problems similar to those involved in cloning cyclic structures. In Figure 8, we would like both $\mathtt{A} =^{\infty} \mathtt{B}$ and $\mathtt{C} =^{\infty} \mathtt{D}$ to be true, but it is not obvious how to arrive at these results. (The attributes f and b are intended to suggest “forward link” and “backward link” respectively.) Meyer’s technique for deep comparison of objects X and Y is to recursively compare references from X and Y under the assumption that $\mathtt{X} =^{\infty} \mathtt{Y}$. For example, in the deep comparison of A and B in Figure 8, since we have

$$\mathtt{A.b} =^{0} \mathtt{B.b} =^{0} \mathtt{Void}$$

and

$$\mathtt{A.f.f} =^{0} \mathtt{B.f.f} =^{0} \mathtt{Void}$$

we need only show that

$$\mathtt{A.f.b} =^{0} \mathtt{B.f.b}.$$

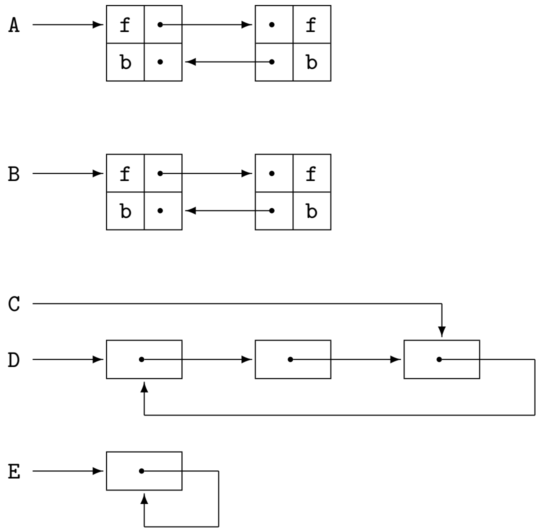


Fig. 8. Deep comparisons

In fact, `A.f.b` $=^0$ `A` and `B.f.b` $=^0$ `B`, and the result follows from the assumption `A` $=^\infty$ `B`. This technique is, of course, analogous to a proof of the correctness of a recursive function in which we use the correctness of the function as an induction hypothesis. Note that even Meyer's definition of deep equality does *not* require structural equality: for instance, `D` $=^\infty$ `E` in Figure 8.

One of the reasons that cycles can occur in a structure is that components of the structure may have "back-pointers" to components that contain them. "Threads" in a threaded tree are an example. For copying or comparing the entire structure, the rules that we have given apply correctly. Problems arise, however, if we try to copy or compare a component of such a structure: for example, a leaf of a threaded tree. In such situations, it is not feasible to provide general methods, and we must resort to class-specific methods, as described in Section 5 below.

3.4 Operations Across Classes

As mentioned in Figure 3.1 and Figure 3.2, both replacement and comparison are more complicated if the objects belong to different classes. Suppose that the objects are `X` and `Y` and that their classes are C_X and C_Y, respectively. There are four cases to consider for `replace(X, Y)`.

C_Y is a subclass of C_X. In this case, `Y` will in general have at least the attributes of `X`, and possibly additional attributes. Copying from `Y` to `X` is safe in the sense that all of the attributes of `X` will be defined, but information in the additional attributes of `Y` will be lost ("sliced"). This is what happens by default in C++ and some other languages. The abstract value of the object may thus not be conserved; a better solution would be to require an explicit slicing operation (i.e., a projection) before the assignment.

C_X is a subclass of C_Y. Copying from `Y` to `X` is undesirable and not allowed directly in any well-known language, because the additional attributes of `X` will be left undefined (retain their old values). Such copying is possible in C++ if the object `X` is just addressed through a reference of type C_Y.

C_Y and C_X have a common ancestor class C_Z. In this case, only the attributes inherited from C_Z will be copied. This is still less likely to make sense than the previous case, of course. Even this kind of copying will happen in C++ if both objects are addressed through a reference of type C_Z.

C_X and C_Y are not related by inheritance. It may be feasible to define sensible copying and comparison operations, but this will have to be done by the programmer.

Similar considerations apply to comparison: only the attributes common to the classes C_X and C_Y will be compared, which will in most cases not give the results that the class designer would like to have. The equality operator is *not* generated automatically in C++, but if it is defined for a class, it will be inherited by all subclasses that do not redefine it.

Evidently, the only simple and easily understandable way to handle replacement between objects of different classes is the principle we chose in Section 3.1: the operation should fail. Thus, `replace` must either give a failure/success code as a return value, or be able to raise an exception.

In comparison, the equivalent simple principle is that objects of different classes can be compared for equality but the result is always `false`. Non-trivial comparison is performed only between objects of the same exact class, so we have a kind of "type-safe covariance" (cf. Section 2.4) for the equality operation.

4 Language Comparison

In this section, we discuss the facilities for copying and comparing provided by several popular or otherwise relevant object-oriented languages. The languages are introduced in alphabetical order.

4.1 BETA

BETA [15], like its precursor Simula [9], does not provide copying or comparing operations automatically. This is a reasonable design decision because, as we have shown, it is not possible to provide the right functions in all contexts.

4.2 C++

C++ leaves most of the work of copying and comparing to the programmer.[6] The compiler provides a default assignment operator and a default copy constructor

[6] Information about C++ was obtained from the ISO/ANSI Draft Standard available at `http://www.cygnus.com/misc/wp/` and elsewhere.

for a class if the programmer does not define these methods. In our terminology, the default assignment operation "replaces" and the copy constructor "clones". (More precisely, the copy constructor initializes an object that has already been created.) Both operations are shallow ("memberwise") and will give rise to memory management problems if the programmer relies on them for classes that contain references. Consequently, C++ programmers are encouraged to define their own assignment operators and copy constructors for classes with reference (pointer) attributes.

C++ provides the operator `==` for built-in types but does not provide a comparison method for user-defined classes. The programmer must provide an overloaded version of the `==` operator for any class whose instances must be compared. The operator `==` with pointer arguments can be used for identity comparisons. Caution is required because, in the presence of multiple inheritance, a C++ object may have several different addresses [18, page 57].

There is a bias in C++ in favour of copying objects rather than sharing them. If data members are not pointers and objects are allocated on the stack, then assignment, argument passing, and returned objects all involve copying. The default assignment operators and copy constructors provided by C++ are consistent with this bias. As classes get more complex, however, complex objects are more likely to be represented as pointer structures. The default methods are inappropriate for pointer structures and programmers must manage copying for themselves. Lalonde and Pugh [11] consider the bias towards copying to be a deficiency of C++ and discuss the implications of this bias for C++ programmers. Style guides for C++ programmers often recommend the use of pointers and heap allocation rather than containment and stack allocation to avoid the overhead and complications of copying [10,19].

4.3 DSM

Although the language DSM [20] is no longer widely used, we include it in this discussion because it is one of the few languages that addresses the issues raised in this paper in a comprehensive manner. Rumbaugh points out that there are a variety of operations that should be propagated to the objects of a structure. These operations include not only `copy` (clone), which we have discussed, but also `save`, `destroy`, `print`, and others. On the other hand, replacing and comparing are not treated because the approach applies only to unary operations.

In DSM, reference attributes *in* classes are replaced by relations (relationships, associations) *between* classes; this approach is quite common in data modelling but less common in programming languages. Propagation is controlled by *propagation attributes* that are part of the semantic relationship. (Obviously, Rumbaugh is using "attribute" in a different sense than that of this paper.) For example, for the class `Car`, the `part-of` relation has the attribute `propagate` and the `made-by` relation has the attribute `shallow` in an example of [20]. Consequently, when we copy a `Car`, the new object will contain copies of the parts of the original `Car` but a reference to the same manufacturer as the original `Car`.

A relation in DSM is always bidirectional, corresponding to a reference attribute and its inverse. In the Car example, the made-by relationship has the propagate attribute in the inverse direction, indicating that copying a manufacturer copies all the cars manufactured by it. We would certainly prefer the none attribute, meaning that the new company must start without any cars manufactured yet. However, our disagreement with Rumbaugh shows only that the semantically correct propagation of cloning cannot be decided automatically by the language or compiler, since even reasonable designers can have different opinions on it.

The DSM principle could obviously be applied also to reference attributes. It is more fine-grained and complicated than our proposal, especially because the attributes are specified separately for each operation. Consequently, a programmer might inadvertently define very strange semantics for a relation. DSM also takes into account the complete graph structure of objects and relations, as we do in the proposals of Section 5.

4.4 Eiffel

Attributes in Eiffel classes are usually represented by references. It is possible to declare an attribute of a class (or a variable in general) as expanded, in which case it will contain an embedded object without separate identity, rather than a reference. It is also possible to declare an entire class as expanded, in which case all variables whose type is that class will contain objects rather than references to objects. Basic types are regarded as special cases of expanded types.

The Eiffel type hierarchy is bounded above by ANY, an ancestor of all classes, and bounded below by NONE, a descendant of all classes. Methods for copying and comparing are defined in class ANY and may be inherited or redefined by user classes. Most of these methods exist in three forms. The basic form is the one that we consider here. In addition to the basic form, there may also exist: a "frozen" form, indicated by the prefix standard_, that cannot be redefined; and a form that allows either or both of its arguments to be Void — that is, a null reference.

In Eiffel [16, 295–307]:

- If X and E have expanded types, the statement X := E is a copy operation.
- If X and E have reference types, the statement X := E is a reference assignment.
- The method is_equal tests for shallow equality. (The operator "=" may be used for the same purpose.)
- The statement X.copy(Y) makes X a shallow copy of Y and ensures that the expression X.is_equal(Y) yields TRUE.
- The expression clone(X) creates a new object, Y; shallow copies the attributes of X to Y; and returns a reference to Y. The (useless) expression clone(X).is_equal(X) yields TRUE.
- The method deep_equal tests for deep equality.

- The methods deep_copy and deep_clone "replicate an entire data structure, starting at the source and creating new objects as needed". Both methods ensure that deep_equal is satisfied. The distinction between deep_copy and deep_clone is not clear in the original specification of Eiffel [16]. Probably, deep_copy was intended to effect a replacement of non-reference attributes and a deep cloning of reference attributes. It is not mentioned at all in later descriptions of the language [17].

After X.copy(Y) or X:=clone(Y) has been executed, the objects X and Y may contain pointers to the same object. After X.deep_clone(Y) or X.deep_copy(Y) has been executed, X and Y are disjoint data structures. Deep operations detect cyclic references and process them appropriately.

Eiffel provides an unusually large collection of methods for copying and comparing, and is the only language we know besides DSM that can handle cyclic structures correctly. Nevertheless, the probability that one of the methods provided would be precisely correct for copying or comparing complex objects seems rather small, and the programmer has no mechanism for incrementally modifying these methods without losing the built-in handling of cyclic structures.

4.5 Java

Java [2] provides single inheritance with the class Object at the root of the class hierarchy. A class other than Object inherits either Object or another class — Java does not provide multiple inheritance for classes — but it may inherit any number of *interfaces*.

The root class Object provides a method clone that creates a new object and then shallow-copies the attributes of the source object to it. A class can support clone in its basic form, redefine it, or prevent its instances from being cloned.

A class that requires a copying method different from clone obtains it by redefining the default implementation. The object must first execute Object.clone, which allocates space for the new object and initializes its attributes as described above, and then overwrite any attributes for which the default values are incorrect.

Although Java does not provide a general method for copying other than clone, some of the standard classes (e.g., Vector and String) provide specialized operations.

Java provides the operator == for identity comparisons. The class Object provides a method equals which defaults to identity comparison. In Java standard classes, equals is redefined to provide an appropriate value comparison. User-defined classes can use the default version of equals or provide an implementation of equals that is appropriate to their needs.

4.6 Smalltalk

A Smalltalk[7] object is *immutable* if its class has no methods that change the values of its attributes. An *identity object* has exactly one instance. For example, a Smalltalk implementation provides an identity object `true` that is never cloned; you are allowed to "copy" `true` but all you will get is a reference to the original object. All identity objects are immutable.

The Smalltalk statement `X := E` evaluates the expression `E`, obtaining an object, and makes `X` a reference to that object. All Smalltalk implementations provide methods `copy` and `shallowCopy`, both of which return a shallow copy of the source object. Some Smalltalk implementations also provide a method called `deepCopy` that returns a new object in which references have been replaced by shallow copies of their referents. Note that this is `clone`^{2}, not a deep copy, in the sense of Section 3.1 of this paper.

Smalltalk provides two comparison operators: "`==`" for identity comparison and "`=`" for equality comparison.

Identity comparison is an ordinary object operation in Smalltalk, which means that it can be redefined in a user class. (It is a Smalltalk programmers' *convention* that users do not redefine "`==`" but the language does not prevent redefinition.) This approach is unfortunate, for two reasons. First, identity is a basic object concept whose meaning should not be changeable [8]. Second, there is a performance penalty for what should be a very rapid test, because object operations are dynamically bound.

5 Semantic Copying and Comparing

The traditional classification of copying and comparing that we have presented — reference, shallow, and deep — is based on the representation of objects rather than on their abstract values. Practical applications frequently require reference operations that are usually provided by the language. The shallow and deep operations provided by the language, if any, are often not suitable in practice. In general, a compiler cannot infer the abstract value of an object from the program text and therefore cannot provide appropriate copy and comparison operations for complex classes. The question that we address is: what facilities should a language provide to simplify the task of defining these operations?

What does it mean, for example, to copy an instance of `GraphicalObject` (see Figure 3)? The target object should presumably share the attribute `shape` of the source object but should have its own attribute `position`. Alternatively, if the target object is to be a member of a group of objects, its position should be that of the group.

[7] Information about Smalltalk was obtained from the FAQ by David N. Smith available at `http://www.dnsmith.com/SmallFAQ/`.

5.1 Attribute Classification

In practice, the distinction between "essential" and "accidental" that we made in Section 2.1 is not always sharp enough to use as a basis for implementation: we need to identify attributes that require special treatment. It seems useful to classify attributes as follows (better terms might be invented):

- structure (composite link, essential object reference);
- normal reference (association) or value;
- accidental reference (association) or value; and
- special attribute.

We give below the equivalent DSM propagation modes (propagation attribute values, see Section 4.3). Note that in DSM these apply only to reference attributes.

The structure attributes (composite links, DSM: `propagate`) are those which are to be followed in both copying and comparing, using standard algorithms for directed graphs to obtain or check graph isomorphism (structure equality). They need not be restricted to point from a composite to its parts; thus directed cycles are possible.

Normal attributes (DSM: `shallow`) are copied or compared as such, in the sense of shallow copying or comparing.

Accidental attributes (DSM: `none`) are not compared at all. In copying, an accidental attribute is assigned a default value if such is given in the class definition. Otherwise, an accidental reference attribute is set to null. Some value *types* may also have natural default values, but, for example, it is questionable to use zero as a default value for all integers.

The special attributes (DSM: `none`) would not be touched by the standard operations. If special attributes are present, the standard operations must invoke class-specific methods to handle them. However, the class-specific methods could access also other attributes and call other methods. Defining them would be allowed even when there are no special attributes.

We can divide the actions of both copying and comparison on an object into three phases:

1. the propagation of the operation over the composite links (if any) to the adjacent objects;
2. handling the normal and accidental attributes; and
3. the class-specific actions.

The execution order between the first part and the other parts is immaterial in cloning and comparing; they could even proceed concurrently. In comparison, the order between the second and third parts is also immaterial, except possibly for performance reasons. In cloning, the order can be important, because the special actions can access also the normal and accidental attributes.

In replacing (and swapping, if that operation is desired), phase 1 must first be propagated to the end, and only if the whole structures are found to be equal are phases 2 and 3 performed for all involved objects. In some cases there might be something that must be done with the old attribute values of the target object. This is straightforward if the language provides garbage collection but can be very awkward otherwise.

5.2 General Proposals

As a partial answer to the problems raised in the preceding sections, we propose the following guidelines.

1. The language should provide:
   - a built-in reference assignment operator; and
   - a built-in identity comparison method.

   The assignment operator allows programmers to create multiple references for a single object — in other words, to introduce aliasing. Although *unintended* aliasing may be harmful, the object model sometimes requires multiple references to an object. The identity comparison enables a programmer to determine whether two names refer to the same object.
2. The language should *not* provide separate public methods for shallow and deep copying and comparison, but only one copy method and one comparison method for each class. The designer of the class, rather than its clients, should choose appropriate semantics for these methods.
3. The language should provide syntactic mechanisms for:
   - distinguishing mutable and immutable classes;
   - distinguishing essential attributes and accidental attributes; and
   - specifying when deep copies or comparisons of reference attributes are required.

   An explicit distinction between mutable and immutable classes enables the compiler to make a number of optimizations. An explicit distinction between essential and accidental attributes enables the compiler to generate the default copy and comparison methods described in item 4 below. The depth specifications enable the programmer to customize these methods.
4. The implementation should provide:
   - an optional default copy procedure for each user-defined class; and
   - an optional default comparison method for each user-defined class.

The copy procedure shallow-copies each essential attribute of the source object, unless the programmer has indicated that an attribute should be deep-copied. The copy procedure does not copy accidental attributes but should provide appropriate default values for accidental attributes in the target object. Similarly, the comparison method shallow-compares essential attributes, unless the programmer has requested deep comparison, but does not compare accidental attributes.

5.3 Applying the Proposals

To illustrate the application of these guidelines, we suggest ways in which they might be incorporated into C++ in a way would fulfill our requirements. We do this to demonstrate the feasibility of our proposals rather than in the realistic expectation of their adoption into C++. In particular, we do not address the more arcane aspects of C++, such as private or protected inheritance. But note that static attributes are irrelevant to copying and comparing and that, for other attributes, it does not matter whether they are public, protected, or private.

Standard C++ already contains some of the features that we need:

- the operator "`=`", used with pointer arguments, provides reference assignment;
- the operator "`==`", used with pointer arguments, provides identity comparison; and
- the keyword `const` can be used to distinguish between mutable and immutable objects (and, in fact, enables mutability distinctions of finer granularity than this).

There are four ways in which we might make the distinctions between essential and accidental attributes, and between deep and shallow operations.

1. We could introduce new keywords. This is incompatible with C++ style, which prefers to overload existing keywords (e.g., `static`) in order to avoid breaking old code.[8] Another disadvantage of new keywords is that the compiler must be modified.
2. We could introduce a convention for marking variable names. For example, we could use the prefix `a_` to indicate an accidental attribute. We could also use the prefix `e_` to indicate an essential attribute, although this would be redundant.
   Similar conventions have been proposed for other purposes. For example, Lakos [10, page 91] suggests using the prefix `d_` for class data members and `s_` for static members. The problem with conventions, however, is that the compiler will not act on them.
3. We could use pragmas to make the distinctions. Pragmas are less offensive than keywords[9] and a compiler is allowed to ignore a pragma that it does not recognize.
4. We could design arbitrary extensions and use a preprocessor to translate the extended language into C++.

We are currently applying the fourth approach, using a preprocessor for C++. The preprocessor, which is based on earlier work [4] and will be described in detail elsewhere [6], performs a number of tasks of which the following are relevant for this discussion:

[8] "Proposing a new keyword . . . . never fails to cause a howl of outrage" [21, page 152].

[9] But: "Too often, `#pragma` seems to be used to sneak variations of language semantics into a compiler and to provide extensions with very specialized semantics and awkward syntax" [21, page 425].

```
class Detector
    public void startPump () { ... }
    deep Counter *counter;
    long startTime;
    accidental Pump *pump;
    accidental Clock *clock;
```

Fig. 9. Preprocessor input for class `Detector`

- Programmers can indicate that attributes of a class are `deep` or `accidental` in the sense of Section 2.1 of this paper.
- The preprocessor constructs default copying and comparing methods that follow the conventions that we have described with respect to normal, accidental, and deep attributes.
- Programmers can override the actions of the copying and comparing methods for special attributes.
- The preprocessor supports multiple, non-virtual inheritance. Virtual inheritance can be supported but we do not describe the (somewhat messy) details here.

Figure 9 shows the class `Detector` of Figure 1 as it would be presented to the preprocessor. We have included a method, `startPump`, to show how public features are declared; in practice, of course, there would be other functions, including a constructor and a destructor. The listing also shows the syntax for distinguishing shallow/deep attributes and essential/accidental attributes; there are no keywords for "private", "shallow", or "essential" because these are defaults.

First, we describe the way in which the preprocessor handles the qualifiers `deep` and `accidental`. Unqualified attributes are shallow-copied and shallow-compared. For `deep` attributes, the structure graph is explored until either non-reference attributes or reference attributes not marked `deep` are encountered. Attributes marked `accidental` are ignored by default comparison methods; the copying methods set copied attributes to a suitable default value, such as `NULL`.

Second, we describe the methods generated by the preprocessor for a general class. Figure 10 shows the file generated for a class `Z` with two parents, `X` and `Y`.[10] The classes `HAX` and `HAY` are the highest ancestors of `X` and `Y`, respectively. These could be the same class (fork-join inheritance) but, in general, both `X` and `Y` could have more than one highest ancestor. The preprocessor generates functions as required. In Figure 10, the user has provided none of the functions and the preprocessor has generated a complete set. However, if the user had provided `operator=`, the preprocessor would not have generated `operator=` and would have generated `defaultReplace` only if the user's `operator=` called it. The defaults for comparison are similar. Note that objects of different classes are never considered equal. We have not yet addressed the issue of cyclic structures.

[10] The preprocessor actually generates a header file, a definition file, and a documentation file. It also includes more white space than shown here.

```
class Z : public X, public Y {
public:
    virtual Z * clone();
    virtual Z & operator=(Z & other);
    virtual bool operator==(const HAX * other) const;
    virtual bool operator==(const HAY * other) const;
private:
    int iz;
    virtual Z & defaultReplace(Z & other);
    virtual bool defaultEqual(const Z * other) const;
    };
Z * Z::clone() {
    Z *result = new Z;
    *result = *this;
    return result;
    }
Z & Z::defaultReplace(Z & other) {
    ( (X &) *this ) = other;
    ( (Y &) *this ) = other;
    iz = other.iz;
    return *this;
    }
Z & Z::operator=(Z & other) {
    return defaultReplace (other);
    }
bool Z::operator==(const X * other) const {
    return
        typeid(* this) == typeid(* other) &&
        defaultEqual((Z *)other);
    }
bool Z::operator==(const Y * other) const {
    return
        typeid(* this) == typeid(* other) &&
        defaultEqual((Z *)other);
    }
bool Z::defaultEqual(const Z * other) const {
    if ( ! ( (X *)this == (X *)other ) ) return false;
    if ( ! ( (Y *)this == (Y *)other ) ) return false;
    if (iz != other->iz) return false;
    return true;
    }
```

Fig. 10. Declaration and implementation of class Z

It is not possible to decide statically whether a structure may contain cycles, but the assumption that every structure might be cyclic introduces considerable overhead because mark bits are required. In the general case, a global view of the object structure is needed. Such a view can in principle be built on an existing language if it has sufficiently powerful reflective facilities, as does Smalltalk (although it is surprising that the standard methods of Smalltalk are so defective in this respect). In the static variety of object oriented languages, the basic mechanisms must be built-in, as they are in Eiffel but in no other well-known language. It should be possible to define incremental methods to adjust the copying and comparison to the exact semantics required for specific classes. Our attribute classification scheme partly solves the problem of cyclic references because only cycles consisting of *deep* references are relevant, and these should not be very common. If deep references are used only for composition, there will be no directed cycles. If sharing of parts is not allowed, there will not even be undirected cycles.

6 Conclusion

Copying and comparing are operations that cannot be generated automatically from a syntactic object description. Nevertheless, most object oriented programming languages provide some support for copying and comparing, either in the form of default methods in root classes, or in some other way.

The subtleties involved in copying and comparing non-trivial objects are such that simple-minded attempts by the compiler to provide suitable methods are likely to be of little use. Consequently, some languages provide only a basic set of methods and leave the rest of the work to programmers.

We have shown that copying and comparing methods for a class can be generated automatically if the compiler or preprocessor is given a few hints about the way in which the attributes of the class are used. In large systems with thousands of classes, automatic generation of these methods could save a considerable amount of work.

The correct handling of object structures that are not simply trees is impossible or at least very cumbersome (depending on the language) to program on the class level. Therefore, this facility should be offered by the language (or standard libraries), but with suitable hooks for the class-specific handling of some attributes.

Acknowledgments. Peter Grogono's part of the research described in this paper was supported by the Natural Sciences and Engineering Research Council of Canada. A significant part of Markku Sakkinen's work was performed at the Department of Computer Science and Information Systems, University of Jyväskylä.

References

1. Serge Abiteboul and Jan Van den Bussche. Deep equality revisited. In T.W. Ling, A. O. Mendelzon, and L. Vieille, editors, *Deductive and Object-Oriented Databases: Fourth International Conference, DOOD '95, Singapore, December 4-7, 1995, Proceedings*, number 1013 in LNCS, Berlin and Heidelberg and New York, 1995. Springer-Verlag.

2. Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, second edition, 1998.
3. Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language: User Guide.* Object Technology. Addison Wesley, 1999.
4. Peter Grogono. Issues in the design of an object oriented programming language. *Structured Programming*, 12(1):1–15, January 1991.
5. Peter Grogono and Patrice Chalin. Copying, sharing, and aliasing. In *Colloquium on Object Orientation in Databases and Software Engineering (ACFAS'94)*, Montreal, Quebec, May 1994.
6. Peter Grogono and Markku Sakkinen. A view and interface generator for C++. Technical report, Department of Computer Science, Concordia University, November 1999.
7. Sonya E. Keene. *Object-Oriented Programming in COMMON LISP.* Addison-Wesley, 1989.
8. Setrag N. Khoshafian and George P. Copeland. Object identity. In *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 406–416, September 1986.
9. Bjørn Kirkerud. *Object-Oriented Programming with Simula.* Addison Wesley, 1989.
10. John Lakos. *Large-Scale C++ Software Design.* Professional Computing Series. Addison-Wesley, 1996.
11. Wilf LaLonde and John Pugh. Complexity in C++: A Smalltalk perspective. *J. Object-Oriented Programming*, 8(1):49–56, March/April 1995.
12. Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual.* Number 114 in LNCS. Springer-Verlag, Berlin and Heidelberg and New York, 1981.
13. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.
14. Bruce J. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12):70–79, December 1983.
15. Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language.* ACM Press/Addison Wesley, 1993.
16. Bertrand Meyer. *Eiffel: the Language.* Prentice Hall International, 1992.
17. Bertrand Meyer. *Object-oriented Software Construction.* Object Oriented Series. Prentice Hall, second edition, 1997.
18. Scott Meyers. *Effective C++.* Addison-Wesley, 1992.
19. Steven P. Reiss. *A Practical Introduction to Software Design with C++.* Wiley, 1999.
20. J. Rumbaugh. Controlling propagation of operations using attributes on relations. In N. Meyrowitz, editor, *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, pages 285–296, September 1988.
21. Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994.

Developing Security Systems in the Real World

Li Gong

Sun Microsystems

Abstract. The debut of Java technology in 1995 was a significant event for the computer security field. First, the claim that ”Java is secure” immediately attracted the intense scrutiny by the computer security research community. Numerous security bugs were later found (and fixed). Second, Java and the Internet once again highlighted the issue of mobile code security and the need for a comprehensive solution – the original ”sandbox” security model for Java is not sophiscated enough for many applications that Java programmers would like to develop. All these put tremendous pressure on the subsequent commercial releases of the Java Development Kit, where any new security solution must be technically sound and also meet commercial needs (time to market, backward compatibility, solving a real problem while maintaining simplicity and extensibility, etc.) This talk covers my experience and lessons learned during the development of JDK 1.1 and JDK 1.2 that might be useful to those who are bridging the gap between academic research and commercial product development.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, p. 251, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction*

Patrick Th. Eugster[1], Rachid Guerraoui[1], and Joe Sventek[2]

[1] Swiss Federal Institute of Technology, Lausanne
[2] Agilent Laboratories Scotland, Edinburgh

Abstract. Publish/subscribe is considered one of the most important interaction styles for the explosive market of enterprise application integration. Producers publish information on a software bus and consumers subscribe to the information they want to receive from that bus. The decoupling nature of the interaction between the publishers and the subscribers is not only important for enterprise computing products but also for many emerging e-commerce and telecommunication applications.
It is often claimed that object-orientation is inherently incompatible with the publish/subscribe interaction style. This flawed argument is due to the persistent confusion between object-orientation as a modeling discipline and the specific request/reply mechanism promoted by CORBA-like middleware systems. This paper describes object-oriented abstractions for publish/subscribe interaction in the form of Distributed Asynchronous Collections (DACs). DACs are general enough to capture the commonalities of various publish/subscribe interaction styles, and flexible enough to allow the exploitation of the differences between these flavors.

Keywords: Abstraction, concurrency, distribution, asynchrony, publish/subscribe, collection

1 Introduction

This paper presents *Distributed Asynchronous Collections (DACs)*: object-oriented abstractions for expressing different publish/subscribe interaction styles and *qualities of service (QoS)*.

Motivation. With the emergence of wide area networks, the importance of flexible, well-structured and also efficient communication mechanisms is increasing. Basing a complex interaction between multiple hosts on individual *point-to-point* communication models is a burden for the application developer and leads to rather static and limited applications. In mobile communications furthermore, it may not be simple for an application to spot the exact location of a component at any moment. Also may the number of entities interested in certain information vary throughout the entire lifetime of the system. All these constraints visualize

* This work is partially supported by Agilent Laboratories and Lombard Odier & Co.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 252–276, 2000.
© Springer-Verlag Berlin Heidelberg 2000

the demand for more flexible communication models, reflecting the dynamic nature of the applications. The *publish/subscribe* interaction style has proven its ability to fill this gap [22]. Indeed the *decoupling* of parties in *time* as well as *space* is a key to scalability.[1]

Publish/Subscribe Interaction. The classical *topic-based* or *subject-based* publish/subscribe involves a static classification of the messages by introducing group-like notions [26], and is incorporated by most industrial strength solutions, e.g., [7,32]. Publish/subscribe is frequently based on a *push model*, where producers feed a software bus with information and the information is pushed towards consumers from there. Other approaches to "messaging" furthermore integrate *pull-style* mechanisms [24], i.e., consumers actively poll for new information. The term *queueing* is frequently applied when referring to such pull-style interaction. Queueing usually expresses *one-for-all* semantics, which means that one single consumer will consume an information. In contrast, with push-style interaction, an information is generally pushed to all consumers (*one-for-each*). As noticed in [28], some applications need only one interaction style while others require both. Instead of bringing all these variants to a common denominator, much emphasis is usually put on their differences.

Object-Oriented Publish/Subscribe: Does it Make Sense? It is often claimed that "objects" cannot really support the requirements of a publish/subscribe middleware [18]. That argumentation is also commonly used by the promoters of so-called "messaging systems", which claim that objects do communicate through synchronous method invocations which force the interacting parties to be both coupled in time *and* in space.

This paper makes a case against this argument by making an attempt to unify the diverging flavors of publish/subscribe. The argument against a fusion of object-orientation and the publish/subscribe communication style indeed might apply to the current commercial practices in distributed object-oriented computing, which are mainly based on derivatives of the *remote procedure call* (DCOM [25], Java RMI [31], CORBA [23]).[2] As we will convey in this paper, decoupling publishers and subscribers can be made very practical in an object-oriented setting, and the integration of object-oriented principles and messaging can go further than simply wrapping a messaging system with an object-oriented API.

Publish/Subscribe Abstractions. To capture the various interaction styles of publish/subscribe, we propose an abstraction called *Distributed Asynchronous Collection (DAC).* A DAC differs from a conventional collection by its distributed nature and the way objects interact with it: besides representing a collection

[1] Time decoupling: the interacting parties do not need to be up at the same time. Space decoupling: the interacting parties do not need to know each other.

[2] Much effort is currently made to integrate messaging into existing middleware solutions, as shown by [12,24].

of objects (*set*, *bag*, *queue*, etc.), a DAC can be viewed as a publish/subscribe engine of its own. In fact, when querying a DAC for objects, the client expresses its interest in such objects. In other words, the invocation of an operation on a DAC expresses the notion of *future notifications* and can be viewed as a subscription. According to the terminology adopted in the *observer design pattern* [9], the DAC is the *subject* and its client is the *observer*. This abstraction allows to unify different publish/subscribe styles in a single framework, which can be seen as an extension of a conventional collection framework. We will show in this paper how this approach allowed us to mix push and pull models, *one-for-all* and *one-for-each* semantics, along with different *QoS*.

In short, within all publish/subscribe interaction styles none is clearly better than the others for all application purposes. In this paper we present simple abstractions for publish/subscribe interaction, called Distributed Asynchronous Collections. On the one hand, DACs allow to capture the different styles without blurring their respective advantages. On the other hand, DACs unite these styles inside a single framework.

Roadmap. The remainder of this paper is organized as follows. Section 2 recalls the various interaction styles in distributed computing and motivates the need for a subscription-like way of communicating. Section 3 gives an overview of the DAC abstraction. Section 4 gives the basic DAC API, whereas Section 5 presents some preliminary class implementations. In Section 6 we show a simple example of programming with DACs. Section 7 discusses some performance issues of our implementation, and Section 8 contrasts our efforts with related work. Finally Section 9 summarizes our work and concludes the paper.

2 Publish/Subscribe: Commonalities and Variations

Before describing our DAC abstraction, we first overview the basics of publish/subscribe interaction styles. In a first step, the publish/subscribe communication style is compared with more traditional interaction schemes. In a second phase, the different existing approaches to publish/subscribe are elucidated more precisely. We point out the fact that each of the different interaction styles has proven certain advantages over others, which motivates the usefulness of unifying them inside a framework.

2.1 Publish/Subscribe in Perspective

The publish/subscribe paradigm is a loose communication scheme for modeling the interaction between applications in distributed systems. Unlike the classic *request/reply* model or *shared memory* communication, publish/subscribe provides *time decoupling* (i.e., the interacting parties do not need to be up at the same time) of message producers and consumers. Figure 1 shows a comparison of the most common communication schemes: *message passing* (*asynchronous* *send*) may also offer an asynchronous interaction scheme, but lacks *space decoupling* (i.e., the interacting parties need to know each other), just like the request/reply communication style. Indeed with message passing, the information producer must have a means of locating the information consumer to which the information will be sent, whereas with the request/reply interaction model the message consumer requires a reference to the information producer in order to issue a request to it. Publish/subscribe combines *time* as well as *space* decoupling, since the information providers and consumers remain anonymous to each other. This outlines the general applicability of this communication model and makes it appealing.[3] Like communication based on shared memory, publish/subscribe moreover allows to address several destinations (*arity of n*). Basically the publish/subscribe terminology defines two roles:

- *Subscriber*: A party which is interested in certain information (events, messages) subscribes to that information, signalling that it wishes to receive all pieces of information (event notifications, messages) manifesting the specified characteristics. *Leasing* is a special form of subscribing, in which the duration of the subscription is limited by a time-out.
- *Publisher*: A party that produces information (events, messages) becomes a *publisher.*

In most applications however, participating entities incorporate both publishers and subscribers, which allows a very flexible interaction. This is one of the main differences to pure *push-based systems* [13], where participants are either producers or consumers and producers are supposed to be several orders of magnitude higher in number than consumers.

	Time	Space	Arity
Request/Reply	Coupled	Coupled	1
Asnynchronous Send	Decoupled	Coupled	1
Shared Memory	Coupled	Decoupled	n
Publish/Subscribe	Decoupled	Decoupled	n

Fig. 1. Different Communication Models

2.2 Topics

The classic publish/subscribe interaction model is based on the notion of *topics* or *subjects*, which basically resemble groups [26]. Subscribing to a topic T can be viewed as becoming member of a group T. The topic abstraction however

[3] It is possible to build closer coupled communication models on top of loose ones and vice versa, as proposed by [34] for instance. The resulting performance in the second case however is generally poor.

differs from the group abstraction by its more dynamic nature. While groups are usually disjoint sets of members (e.g., group communication for replication [3]), topics typically overlap, i.e., a participant subscribes to more than just one topic. In order to classify the topics more easily, it is of great use to furthermore introduce a hierarchy of topics [32]. In this model, a topic can be a derived or more specialized topic of another one, and is therefore called *subtopic*. The use of wildcards offers a more convenient way of expressing *cross-topic* requests.

Figure 2 shows an example of topic-based subscribing. Subscriber S_1has announced its interest in both topics x and y. It is notified of events corresponding to both topics (messages m_x and m_y). Subscriber S_2 has only subscribed to topic x, and therefore only receives messages related to that topic (message m_x).

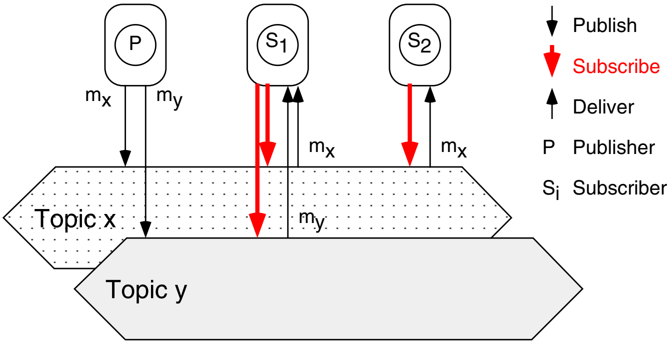


Fig. 2. Topic-Based Subscribing

2.3 Push and Pull Mixing

In the publish/subscribe model, the action of subscribing describes a sort of registration procedure for an interested party. However, interests in events can also be expressed through a more direct interaction. In general, we distinguish the following ways for an interested party to interact:

- In a passive way, it can subscribe to a choice of notifications. By callbacks it will be notified of the occurrence of events. This kind of interaction constitutes the *push model*, since the information is pushed from the information bus to the subscriber. This is the classic publish/subscribe approach, since it enforces applications which are only loosely coupled in *time*.
- More actively, a consumer can *poll* for new notifications. This task may waste resources and is not well adapted to asynchronous systems. In fact, polling based solutions tend to be very expensive and scale poorly: polling too often can be inefficient and polling too slowly may result in delayed responses to critical situations [29]. This style is often called *queueing*, and in the context of this paper this is the type of interaction we will refer to as *pull model*.

– Another yet more synchronous pull-type interaction is given by *blocking* pull-style interaction. In this scenario, a consumer which tries to pull information is blocked until a new notification is available. Just like the request/reply model however, this variant introduces time coupling, and is therefore rarely used in common messaging systems.

Although in general push-style interaction seems more appropriate, certain applications may not be interested in receiving information as soon as possible, but only at precise moments. In those situations, the pull model might be of interest.

2.4 Delivery Semantics and Reliability Issues

In distributed systems, and in particular when considering communication models and protocols, precise specification of the semantics of a delivery is a crucial issue. Delivery guarantees are often limited by the behavior of deeper communication layers, down to the properties of the network itself, limiting the choice of feasible semantics. On the other hand, different applications also may demand for different semantics. While sometimes a high throughput is preeminent and a low reliability degree is tolerable, some applications prioritize reliability to throughput. For this reason most common messaging systems provide different *qualities of service*, in order to meet the demands of a variety of application purposes [1,32].[4] The delivery semantics for notifications offered by existing systems can be roughly divided into two groups.

– *Unreliable delivery.* Protocols for unreliable delivery give few guarantees. These semantics are often used for applications where the throughput is of primary importance, but the loss of certain messages is not fatal for the application.
– *Reliable delivery.* Reliable delivery means that a message will be delivered to every subscriber despite certain failures. Usually the failure or the absence of the subscriber itself is not considered, i.e., if the subscriber has failed, the message might not be delivered to it and the reliability property is not considered violated. When using persistent storage to buffer such messages until the subscriber is back on line, a stronger guarantee is given. This is often referred to as *certified delivery* [32].

3 Distributed Asynchronous Collections: Overview

This section gives an overview of our approach to publish/subscribe, by first introducing *Distributed Asynchronous Collections* as key abstractions. We show the relationship between those abstractions and the publish/subscribe communication model. In a second step, we picture more in detail how these abstractions allow to build several different publish/subscribe variants inside a unified framework. This section however should be understood as a general introduction to

[4] [32] adopts the notion of *delivery service*.

our abstractions for publish/subscribe. The following sections will give a more concrete view of DACs.

3.1 DACs as Object Containers

Just like any collection, a DAC is an abstraction of a container object that represents a group of objects. It can be seen as a means to store, retrieve and manipulate objects that form a natural group, like a mail folder or a file directory. Unlike a conventional collection, a DAC is a distributed collection whose operations might be invoked from various nodes of a network. DACs differ fundamentally from the distributed collections described in [21] for instance, by being asynchronous and essentially distributed, i.e., DACs can be seen as omnipresent entities.[5] Participating processes act with a DAC through a local proxy, which is viewed as a local collection and hides the distribution of the DAC. DACs are not centralized on a single host, in order to guarantee their availability despite certain failures.

A collection framework is a unified architecture for representing and accessing collections, allowing them to be manipulated independently of their representation. For example, both Smalltalk [14] and Java [16] contain rich collection frameworks that reduce the programming effort by providing useful data structures and algorithms together with high-performance implementations. Collection frameworks can for instance also be found for C++ (e.g., Silicon Graphics' STL [30]) as additional libraries. Figure 3 shows the inheritance graph of the Java collection framework.

3.2 The Asynchronous Flavor of DACs

Our notion of Distributed Asynchronous Collection represents more than just a distributed collection. In fact, a synchronous invocation of a distant object can involve a considerable latency, hardly comparable with that of a local one. In contrast, asynchronous interaction is enforced with our collections. By calling an operation of a DAC, one expresses an interest in *future notifications.* When querying a DAC for objects of a certain kind for instance, the party interacting with the DAC expresses its interest in such objects. Therefore, when such an object is eventually "pushed" into the DAC, the interested party is asynchronously notified.

There is a strong resemblance with the notion of *future* [4] (*future type message passing* [35]), that describes a communication model in which a client queries an *asynchronous object* for information by issuing a request to it. Instead of blocking however, the client can pursue its processing. As soon as the reply has been computed, the object acting as server notifies the client. Latter one may query the result (*lazy synchronization* or *wait-by-necessity* [5]), or ignore it. Figure 4 compares the two paradigms. When programming with DACs, the

[5] The distributed collections presented in [21] are centralized collections that can be remotely accessed through RMI.

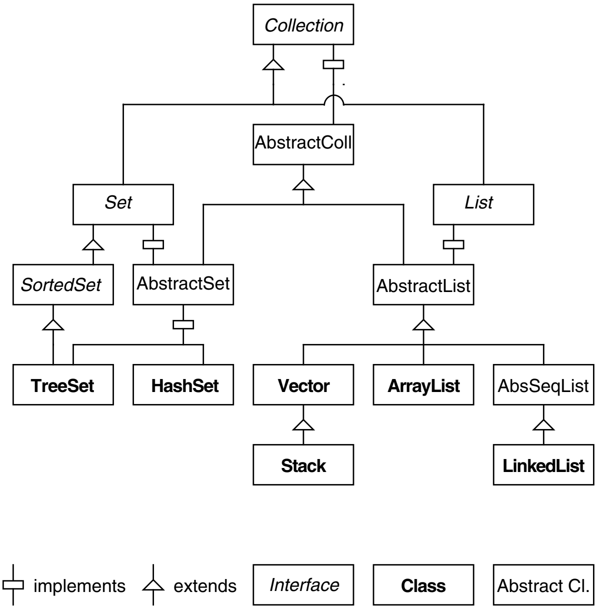


Fig. 3. Collections in Java (Excerpt)

subscriber can be viewed as the client. The DAC incarnates a server role in this scenario, since the publishers, which are the effective information suppliers, remain anonymous.

By calling an operation on the DAC, the caller requests certain information. The main difference with futures lies in the number of times that information is supplied to the client. Within the notion of future, only a single reply is passed to the client,[6] whereas with DACs, every time an information which is interesting for the registered party is created, it will be sent to it.

3.3 Publish/Subscribe with DACs

Expressing ones interest in receiving information of a certain kind can be viewed as subscribing to information of that kind. By viewing event notifications as objects, a DAC can be seen as an entity representing related event notifications. Clearly, if a collection is a set of somehow related objects, a DAC can be seen as a set of related "events". When considering the classical topic-based approach to publish/subscribe, a DAC can be pictured as an extension of a conventional collection but also as a representation for a topic. It is always possible to insert a new element into a DAC. In the sense of publish/subscribe, inserting an object

[6] ABCL/1 represents an exception, in the sense that several replies may be returned [35].

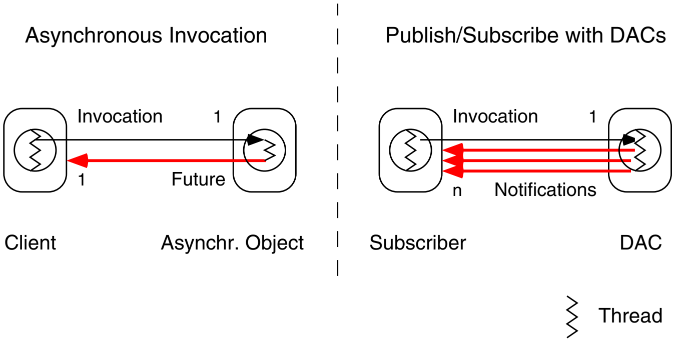


Fig. 4. DACs vs. Future

into a DAC also means to publish that object for the topic represented by the DAC. Every DAC can thus be viewed as a publish/subscribe engine of its own. Figure 5 shows the traditional topic-based publish/subscribe scheme. The topic is represented by an attribute of the message, and the application has to deal with it explicitly. Since a DAC is bound to a topic, the topic is given implicitly, and appears only in the protocol message which is hidden from the application, as shown in Figure 6. It encapsulates the application message.

Existing publish/subscribe frameworks introduce specialized message types, e.g., [12]. Our approach frees the application programmer from the burden of marshalling and unmarshalling data into and from dedicated messages. In our context, a message can be basically of any kind of object. In Java, this is expressed by allowing any object of class `java.lang.Object` to be passed as a message.[7]

Message m	`public class Message {`
	`  public String topic;`
	`  public String content;`
	`}`

Criteria	topic of m is “/Chat/Insomnia”
Argument	`String topic = "/Chat/Insomnia"`
Evaluation	`m.topic.equals("/Chat/Insomnia")`
Deliver	`m`

Fig. 5. “Traditional” Topic-Based Publish/Subscribe

[7] In order to be conveyable, a Java object should furthermore implement the `java.io.Serializable` interface [15], which contains no methods.

Protocol Message p	`public class Message {`
	`  public String getTopic() {...}`
	`  public Object getMsg() {...}`
	`  ...`
	`}`

Message m	`public class ChatMsg {...}`
Criteria	topic of m is “/Chat/Insomnia”
Argument	`String topic = "/Chat/Insomnia"`
Evaluation	`p.getTopic().equals("/Chat/Insomnia")`
Deliver	`m = p.getMsg()`

Fig. 6. Topic-Based Publish/Subscribe with DACs

4 DAC Interfaces

The previous section introduced DACs as general abstractions for publish/subscribe. This section presents the main interfaces of our DAC realization in Java. In the context of this paper, we will limit ourselves to describing the functionalities which are common to all DAC subinterfaces, in order to show their similarity to operations on conventional centralized collections.

4.1 Topic-Based Subscribing with DACs

In our system, each *topic* is represented by a DAC, and is denoted by a name, like “Chat”. A DAC constructor thus requires an argument which denotes the name of the topic it will represent (see Section 6 for an example). Topics can have specializations, or *subtopics*, and connecting to a topic requires the name in a URL-type format. Typically, “/Chat/Insomnia” is a reference to the topic called “Insomnia” which is a subtopic of “Chat”. The root of the hierarchy is represented by an *abstract* topic (denoted by “/”). Top-level topics, which are no specializations of already existing ones, are subtopics of the abstract root topic only. Subscribing to a topic can trigger subscriptions for the subtopics as well, as illustrated in Figure 8. Subscriber S_1 subscribes to topic “Chat” and claims its interest in all subtopics. Hence S_1 does not only receive message m_2 but also message m_1 published for topic “/Chat/Insomnia”. In contrast, S_2 only subscribes to “/Chat/Insomnia” and thus does not receive message m_2, which belongs to the *supertopic*. With the *push* model adopted in DACs, subscribing entities must register a *callback* object. That callback object must implement a specific interface, namely the `Notifiable` interface, shown in Figure 7. Through a call to the `contains()` method, the DAC notifies the subscriber that it contains a new notification. The second argument enables the use of the same callback object for several topics.

```
public interface Notifiable {

  public void contains(Object msg, String topicName);
}
```

Fig. 7. Interface Notifiable

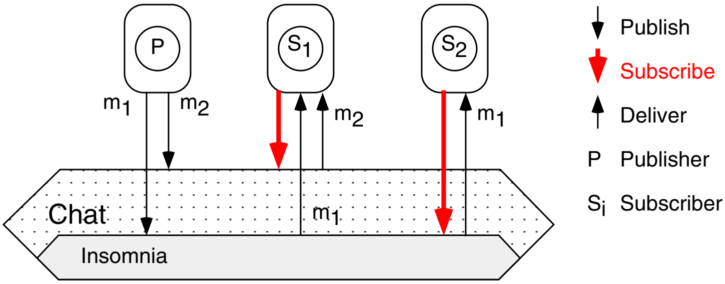


Fig. 8. Topic-Based Publish/Subscribe with DACs

4.2 DAC Methods

Figure 9 summarizes the main methods of the base DAC interface. More sophisticated interfaces like the `DASet` all derive from this interface, but are omitted for the sake of brevity. We roughly distinguish *synchronous* and *asynchronous* methods.

Synchronous Methods. Since a *DAC* is in the first place a collection, the `DAC` interface inherits from the standard `java.util.Collection` interface. The inherited methods are not denatured but adapted, and we denote them as *synchronous.*

- `get()`. Similarly to a centralized collection, calling this method allows to retrieve objects. The synchronization it introduces is however very weak, since it returns `null` in the absence of unconsumed information, as explained in Section 2. Which element will be returned depends on the nature of the collection (see Section 5 for more details). This implements the *pull* model.
- `contains()`. A DAC is first of all a representation of a collection of elements. This method allows to query the collection for the presence of an object. Note that an object that is contained in a DAC belongs to the topic represented by that DAC.
- `add()`. This method allows to add an object to the collection. The corresponding meaning for a DAC is straightforward: it allows to publish a message for the topic represented by that collection. An asynchronous variant of this method could consist in advertising the eventual production of notifications.

This could furthermore be combined with the registration of a callback object, that the DAC would poll in order to obtain new event notifications. In the terminology adopted in [24], this is called a *pullsupplier*.

Asynchronous Methods. We have added several *asynchronous* methods to express the decoupled nature of publish/subscribe interaction specific to DACs. In these methods, asynchrony is expressed by an additional argument, denoting a callback object which implements the `Notifiable` interface. Not all operations known from conventional collections find an analogous meaning in an asynchronous distributed context, and our ongoing research in that domain might cause minor modifications to this interface.

- `contains(Notifiable n)`. The effect, for instance, of invoking this method is not to check if the collection already contains an object revealing certain characteristics, but is to manifest an interest in any such object, that should be eventually pushed into the collection. The interested party advertises its interest by providing a reference to an object implementing the `Notifiable` interface (Figure 7), through which it will be notified of events.
- `containsAll(Notifiable n)`. This method offers the same signature than the previous method. The difference is that a subscription is generated for all subtopics of the topic represented by this DAC. This reflects the situation given in Figure 8.
- `remove(Notifiable n)`. Likewise, by calling one of these methods, a subscriber does not trigger the removal of an object already contained in the collection, but expresses its interest in being notified whenever an object matching its criteria is inserted in the collection, after which the object will be removed immediately. This expresses that a message is delivered to one single subscriber only. This is frequently called *one-for-all* or *one-of-n* [32] in contrast to *one-for-each*,[8] implemented by the asynchronous `contains()` methods, where a message is sent to all.
- `removeAll(Notifiable n)`. This method is similar to the previous one, except that a subscription is generated for all subtopics of the topic represented by this DAC.
- `clear(Notifiable n)`. The conventional argument-less `clear()` method allows to erase all elements from the collection, whereas this asynchronous variant expresses the action of *unsubscribing*.

5 DAC Classes

The previous section focused on the interfaces, through which an application can use DACs in order to benefit from the strength of our publish/subscribe abstractions. As depicted earlier, our framework consists of a variety of DACs spanning

[8] By using the formalism of [27], one could say that *every Nth occurrence* of an event is notified to a subscriber, with N being the total number of subscribers, and no event being delivered to more than one subscriber.

```
public interface DAC
  extends java.util.Collection

{
  public Object get();
  public boolean contains(Object message);
  public boolean add(Object message);
  ...
  public boolean contains(Notifiable N);
  public boolean containsAll(Notifiable N);
  ...
  public boolean remove(Notifiable N);
  public boolean removeAll(Notifiable N);
  ...
  public void clear(Notifiable N);
  ...
}
```

Fig. 9. Interface DAC (Excerpt)

different semantics and guarantees, since different applications have different requirements. These semantics can be seen as different *QoS*. While certain properties of DACs reflect in their interfaces, certain semantics do not appear in the API. These parameters influence the classes implementing those interfaces, and thus lead to a variety of classes implementing the same interface. This section presents the different properties of the classes constituting our framework.

5.1 Delivery Semantics

When a producer publishes a message, it does not directly interact with subscribers. To whom exactly the message will be delivered does not show in the DACs interface. Parts of the semantics do not come to light in the interfaces. The underlying multicast protocols might lead to different classes implementing the same interface. The `DASet` (Distributed Asynchronous Set) interface, for instance, is implemented by multiple classes. The first one does not offer more than plain unreliable delivery (`DAWeakSet`), whereas others guarantee reliability (e.g., `DAStrongSet`). By distinguishing between unreliable and reliable DACs our framework hierarchy is roughly split into two subtrees, as shown in Figure 10.

5.2 Duplicates

Just like it is possible to have duplicate elements in centralized collections, it is possible in Distributed Asynchronous Collections that a same message is delivered more than once. In fact, the two are closely related in our context. If a DAC does not accept duplicates, it should not deliver any duplicates to subscribers.

The simple DAWeakBag class for instance does not prevent a notification to be delivered more than once, whereas the DAWeakSet class gives stronger guarantees by eliminating duplicate elements. This property is orthogonal to other characteristics of our DACs. For that reason, our framework contains a variant with and without duplicates for every other property, as shown in Figure 10. When allowing duplicates and combining with unreliable delivery for instance, the outcome is *best-effort* semantics. In return, with reliable delivery, *at-least-once* semantics can be guaranteed.

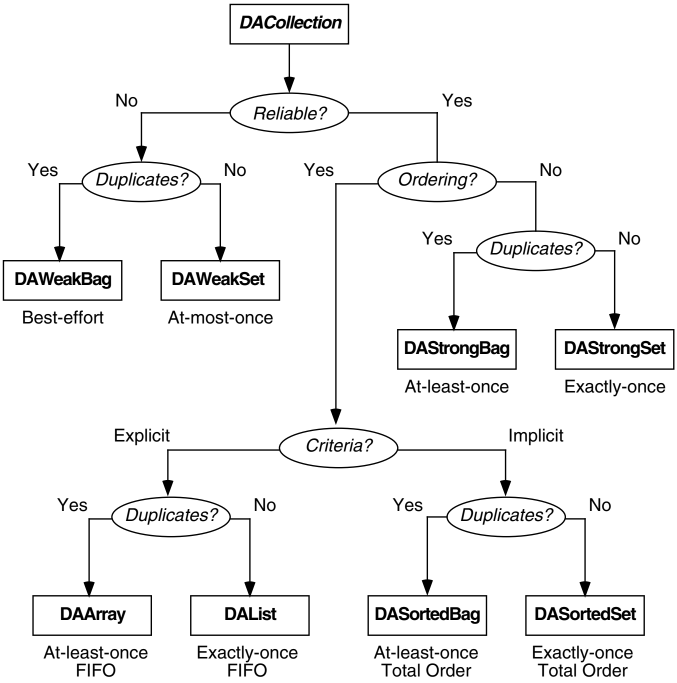


Fig. 10. DAC Framework

5.3 Storage vs. Delivery Order

Collections are often characterized by the way they store their elements. *Sets* or *bags* for instance do not rely on a deterministic order of their elements. Conversely, *sequences* can store their elements in an order given explicitly or implicitly based on properties of the elements. In Distributed Asynchronous Collections however, the notion of space is somehow replaced by the notion of time. If some centralized collections reveal a deterministic storage order, a distributed asynchronous sequence may offer a deterministic ordering in terms of order of delivery to the subscribers. In the Java collection framework for instance, a *sorted set* is a sequence which is characterized by an ordering of the elements based on their properties. This can be seen as an implicit order. With our DACs, an implicit order is a global delivery order on which the DAC itself decides. The `DASortedSet` class for instance presents a *total order* of delivery. Inversely, a *FIFO* delivery order can be seen as an explicit order: it is given by the order in which events are notified to the DAC by a publisher.

5.4 Insertion Order

In different centralized collections, the insertion order may have an impact on the storage order. In a *queue* or a *stack* for instance, the chronological insertion order will drive the storage order as well as the extraction order. A position can be given as additional argument to an insertion into a *list* for instance. In an asynchronous collection however, the order of insertion corresponds to the order of sending or publishing. It seems obvious that inserting an element at a specific position cannot translate to delivering a message at a certain moment in time relative to other messages, since inserting a message at the beginning of a list would translate to sending a message before messages that have possibly already been delivered to subscribers. Therefore there is never any explicit argument for the order passed when "inserting" a new element into a DAC.

5.5 Extraction Order

Extracting an element from a centralized implementation corresponds to pulling messages from a distributed asynchronous one. In the case of consumers polling a DAC for new messages, two different policies may be applied:

- *FIFO.* The collection behaves like a queue by returning the first received and undelivered message. In fact, the DAC proxy contains a buffer, in which received messages are inserted. From there, they are delivered to the pulling consumer in a FIFO order.
- *LIFO.* The collection acts like a stack and delivers the latest received message. The principle is the same than above, except that the messages are delivered in a LIFO order from the buffer to the consumer.

Therefore when using a pull model, the application has the choice between queues and stacks. Any class presented in Figure 10 can be used both as stack or queue.

Messages may be volatile, which means that they may be dropped immediately after delivery. Conversely, the message could be stored in memory or even on persistent storage. In the context of this work however, we did not deal with message storage so far. Messages are considered volatile, and are dropped as soon as they have been consumed. Missed messages are therefore not replayed to *late subscribers* or temporarily disconnected participants.

6 Putting DACs to Work

In this section we describe a simple example application using the flexibility of Distributed Asynchronous Collections. It shows how to implement *chat sessions* based on simple DACs.

We will concentrate on two users, Alice and Tom. They are both chat addicts, and love to chat deep into the night. Therefore they subscribe to the topic "Insomnia" which is a subtopic of "Chat" to receive all messages from like-minded chatters (see Figure 11). For the sake of simplicity, we will assume that this evening Tom is missing inspiration, and therefore takes a pure subscriber role. Alice on the other hand, is very talkative, and publishes several messages. Figure 12 shows class `ChatMsg`, which represents a possible message class for this application.

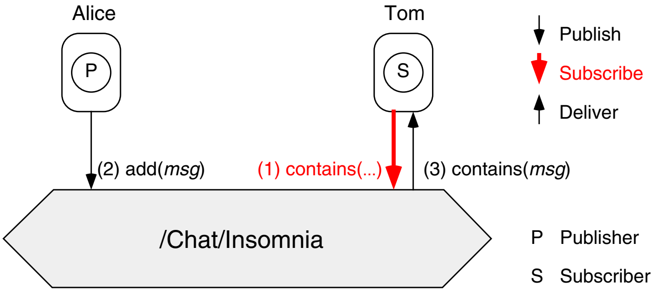


Fig. 11. Chatters

```
public class ChatMsg
  implements java.io.Serializable

{
  private String sender;
  private String text;
  public String getSender() { return sender; }
  public String getText() { return text; }
  public ChatMsg(String sender, String text) {
    this.sender = sender; this.text = text; }
}
```

Fig. 12. Event Class for Chat Example

6.1 Publishing for a Topic

When making use of topic-based publish/subscribe, a topic is represented by a DAC, as seen previously. In order to access a DAC from a process, a proxy must be created. This requires an argument denoting the name of the topic it bears. Except for that argument, the action of creating a proxy is indistinguishable from creating a local collection. The DAC instance called *mychat* in Figure 13 henceforth allows us to access the topic "/Chat/Insomnia". Now it is possible to directly publish and receive messages for the topic associated to that DAC.

Creating an event notification for a topic consists in inserting a message object into the DAC by issuing a call to the `add()` method (see Section 4), from where it is accessible for any party. It is more favorable for consumers to be notified automatically when a new message has been published, than to waste computation time on polling activity. For that purpose, a party interested in a topic can register as subscriber.

```
DASet sleeplessChatters = new DAStrongSet("/Chat/Insomnia");
String me = "Alice";
ChatMsg msg = new ChatMsg(me, "Hi everyone");
sleeplessChatters.add(msg);
```

Fig. 13. Publishing a Message

6.2 Subscribing to a Topic

In order to subscribe to a topic an interested party must provide a callback object implementing the `Notifiable` interface. The callback method comprises two arguments. The first argument represents the effective message, and the second argument represents the name of the topic the message was published for. This provides more flexibility, since the same subscriber object can be used to receive messages related to several topics. In the above example, a subscriber may be interested in all ongoing chat sessions, and not only in "Insomnia". Figure 11 shows the interactions with the DAC, and Figure 14 shows the corresponding code that allows Tom to subscribe.

7 Implementation Issues

This section discusses the realization of our first DAC implementation, including first performance measurements. We draw preliminary conclusions of our prototype, which has been developed in pure Java and relies on UDP, thus increasing its portability.

```
class ChatSubscriber
  implements Notifiable

{
  public void contains(Object msg, String topic) {
    System.out.println(((ChatMsg)msg).getText());
  }
}

DASet sleeplessChatters = new DAStrongSet("/Chat/Insomnia");
Notifiable sub = new ChatSubscriber();
sleeplessChatters.contains(sub);
```

Fig. 14. Topic-Based Publish/Subscribe with DACs

7.1 Inside DACs

The effective DAC class as it is perceived by the application only represents a small portion of the underlying code. Redundant code has been avoided by a modular design and using inheritance. Figure 15 shows the different layers in our implementation. These layers do not necessarily correspond to Java classes, but represent protocol layers.

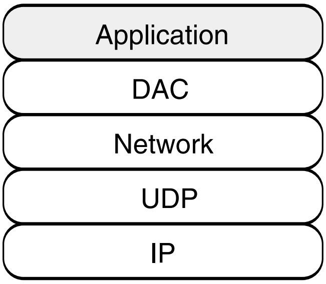


Fig. 15. Layers

– **The DAC layer.** This layer is composed of the classes implementing directly the DAC interfaces. They are rather lightweight classes, which delegate general functionality to the underlying layer. Their tasks are similar to centralized container classes. They mainly take care of the local management of messages, and furthermore handle the subscriptions. The most frequent interaction model is the callback model (push-model), where subscribers do not poll for new messages but are called back upon incoming messages. In

that case the DAC applies a predefined threading model, by assigning notifications to threads.
- **The Network layer.** The Network layer regroups common functionalities of all DACs, like publishing messages or forwarding subscription information. It hides any remote party involved in same topics from the DAC layer. This layer maintains a form of network topology knowledge, which basically consists of its immediate neighbors.
- **The UDP layer.** Our entire publish/subscribe architecture is finally implemented on top of UDP. UDP is a non reliable protocol, which offers us the looseness required for the decoupled nature of publish/subscribe. Java offers classes for UDP sockets and datagrams (`java.net.DatagramPacket` and `DatagramSocket`), which are pretty close to the metal.

7.2 Performance

The performance tests of our prototype were made on HP workstations running HP-UX 10.20 and JVM 1.1.5 and 1.1.6. on a normal working day. The implementation uses a marshalling/unmarshalling procedure built from scratch and optimized for each event type (the Java serialization classes were not used, since they are usually considered rather slow). Four example message types were considered:

- `Integer.` This corresponds to the basic Java `int` type.
- `String.` Java type `String` with a length varying between 10 and 20
- `DetailRecord.` This is a class containing four attributes, of which two represent dates (Java type `Date`) and two are strings (Java type `String`).
- `CallDetailRecord.` A subtype of `DetailRecord`. In addition to the attributes of the latter one, a `CallDetailRecord` furthermore contains four integers and two strings.

In our measurement scenario, several subscribers asynchronously receive events for a topic where a publisher produced the events. The numbers of messages considered for a single run of the experiment varied between 10 and 1000 and the measures obtained conveyed an average result after several experiments of the same profile.

Figure 16 shows the latency when publishing. For example, a publisher needs 3s to publish 100 events of type `DetailRecord`. They include the time for marshalling each of the events and the time to put the events into the UDP socket.

Figure 17 shows the global throughput for the same scenario. It takes for instance 5s until a subscriber has received 100 events of type `DetailRecord`. The 5s correspond therefore to the time spent at the publisher side and the subscriber side of the DAC. They include the time for marshalling, remote communication and unmarshalling.

These simple measurements allowed us to do draw several preliminary conclusions:

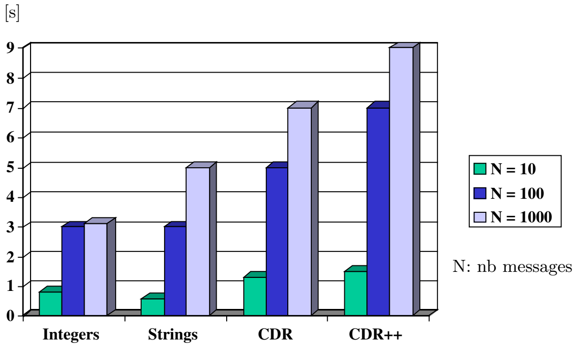


Fig. 16. Latency

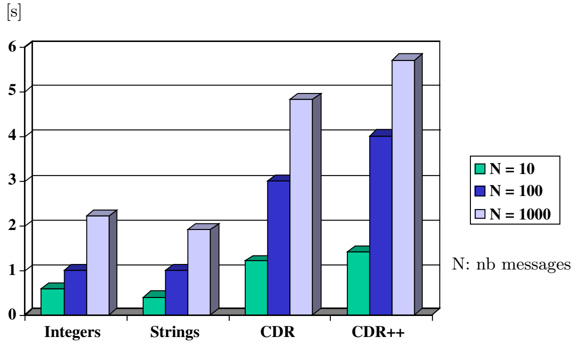


Fig. 17. Throughput

- The complexity of the event type has a heavier impact on the time it takes for a publisher to send events then on a subscriber to receive events. This is not surprising because in the first case, the marshalling time is more significant (there is no inherent cost of remote communication).
- It might look surprising that integers take longer than strings. In this implementation however, everything is converted to strings in the serialization procedure.
- Finally, the overall measures confirm the very fact that nowadays, optimizing marshalling is at least as important as optimizing remote communication.

8 Related Work

During the last years, the need for large scale event notification mechanisms has been recognized. Much effort has therefore been invested in this domain, and a multitude of approaches have emerged from academic as well as industrial impulses. We present here the main characteristics of related approaches and we compare them with our Distributed Asynchronous Collections.

8.1 Event Service Specifications

In order to integrate the publish/subscribe communication style into existing middleware standards, specifications have been conceived by both the Object Management Group [24] and Sun [12,2,6].

The OMG has specified a CORBA service for publish/subscribe oriented communication, called the *CORBA Event Service*. The specification is aimed to be general enough to not preclude sub-specifications and various implementations that would match the needs of specific applications. According to the general service specified however, a consumer subscribes to a channel expressing thereby an interest in receiving *all* the events from the channel. In other words, filtering of events is done according to the channel names, which basically correspond to topic names. When the consumer subscribes to the channel, it is supposed to receive all events put in the channel. Event channels are CORBA objects themselves, and in current implementations they are centralized components. Therefore these engines manifest a strong sensitivity to any component failure, which makes them unsuitable for critical applications.

The *Java Messaging Service* [12] is a specification from Sun. Its goal is to offer a unified Java API around common publish/subscribe engines. Certain existing services implement the JMS, but to our knowledge no publish/subscribe system has been implemented with the goal to merely support the JMS API directly. Its generic nature, required in order to conform to a maximum number of existing systems, appears to be rather cumbersome. Applications are very aware of the underlying messaging service, and developers must make themselves acquainted with the API.

The *Java Distributed Event Specification* [2] explicitly introduces the notion of event *kind*. Registration of interest indicates the kind of events that is of interest, while a notification indicates an occurrence of that kind of event. One can combine this notion with that of *JavaSpace* [8] to provide support for topic-based publish/subscribe notification. Inspired by Linda [10], a JavaSpace is for example a container of objects that might be shared among various suppliers and consumers. The JavaSpace type is described by a set of operations among which a *read* operation to get a copy of an object from a JavaSpace, and a *notify* operation aimed at alerting some potential consumer object about the presence of some specific object in the JavaSpace. Combined with the Java Distributed Event interfaces, one can build a publish/subscribe communication scheme where a JavaSpace plays the role of the event channel aimed at broadcasting events (notifications) to a set of subscriber objects. The nature of the subscription is however not specified and it is not clear whether one would be able to subscribe to a particular operation.

The *InfoBus 1.2 Specification* [6] describes an information bus which enables dynamic data exchange between JavaBeans. Components must implement a minimal interface in order to plug into the bus. As a member of the bus any component can exchange data structured as arrays, tables, or database rowsets with other components. Interestingly, adapted collection types are available for InfoBus, which ease the transfer of collections of objects.

These standards are based on specifications and it would be interesting to see how one could implement services that comply with these standards using DACs. Note however that the CORBA Event Service does not present any hierarchical arrangement of channels, and the Java Messaging Service introduces explicit message classes which require explicit marshalling/demarshalling.

8.2 Established Systems

Most industrial strength solutions involve topic-based publish/subscribe. *Smartsockets* [7] or *TIB/Rendezvous* [32] are such engines.

In Smartsockets, an event channel can accept subscriptions for specific topics. A consumer receives all the event notifications that belong to the topic to which it has subscribed. The topic defines a kind of virtual connector between objects of interest and recipients. If a producer is interested in producing an event on a number of topics or channels, it has to explicitly publish the event on all of them. Event notifications are represented by records, nevertheless custom event types may be defined.

A similar approach was adopted in the development of the TIB/Rendezvous infrastructure. A hierarchical naming model corresponds to the hierarchical organization of the entities of interest. Just as Uniform Resource Locators (URLs) provide a way of locating and accessing Internet resources, a naming scheme is provided to locate and access events of interest. The naming scheme proposed can use wildcards, which allows to subscribe to patterns of topics. TIB/Rendezvous provides a certain degree of fault-tolerance, and makes usage of IP-multicast. Event notifications are composed of a set of typed data fields, including the topic.

Most industrial systems implement API's for object-oriented languages, like the JMS specification for Java. These solutions however did not undergo a fundamentally object-oriented design, but only offer an object-oriented layer on top of a messaging system.

8.3 Collections

Both Java and Smalltalk offer integrated collection frameworks. These only span the most common collection types. More specific collections can be found as external libraries. For instance, *JGL* [21] and the *`util.concurrent`* [20] package offer more elaborate collection types for Java.

JGL is a first approach to distributed collections in Java. It was designed to provide a more advanced series of collections, since the Java environment by default only offers limited support for data collections and algorithms, covering only the main features used by the majority of Java developers. JGL extends the basic Java collections with more refined types. The notion of distributed collection in JGL though describes a centralized collection object, accessible through Java RMI.

The util.concurrent package provides the application programmer with a set of collections especially targeted at resolving concurrency problems. It contains for instance collections which alleviate concurrent traversals by making each time a copy of the array backing the collection. Another feature are synchronization wrappers for standard collections, with the possibility to specify external read and/or write locks.

In contrast to JGL, our DACs avoid any single point of failure and are essentially distributed. To exploit this distribution, asynchronous interaction is enforced. Synchronization as discussed in the context of the util.concurrent package is an issue we do not address with our DACs, but could be the topic of future work.

9 Concluding Remarks

It has long been argued that distribution is an implementation issue and that the very well known metaphor of objects as "autonomous entities communicating via message passing" can directly represent the interacting entities of a distributed system. This approach has been conducted by the legitimate desire to provide distribution transparency, i.e., hiding all aspects related to distribution under traditional centralized constructs. One could then reuse, in a distributed context, a centralized program that was designed and implemented without distribution in mind.

As argued in [33,19,11] however, distribution transparency is a myth that is both missleading and dangerous. Distributed interactions are inherently unreliable and often introduce a significant latency that is hardly comparable to that of a local interaction. The possibility of partial failures can fundamentally change the semantics of an invocation. High availability and masking of partial failures involves distributed protocols that are usually expensive and hard, if not impossible to implement in the presence of network failures (*partitions*).

We have been considering an alternative approach where the programmer would be very aware of distribution but where the ugly and complicated aspects of distribution would be encapsulated inside specific abstractions with a well-defined interface. This paper presents a candidate for such an abstraction: the *Distributed Asynchronous Collection*. It is a simple extension of the well-known collection abstraction. DACs add an asynchronous and distributed flavor to traditional collections [4], and enable to express various forms of publish/subscribe interaction. In fact, most systems we know about are unwieldy and consider only a limited set of interaction models. DACs are general lightweight publish/subscribe abstractions: they can be introduced through a library approach and they allow to express various interaction types and QoS.

We believe that our object-oriented view of publish/subscribe is a unique compromise between transparency and efficiency. By offering a modular design aligned with different communication semantics, we enforce ease of use without missing performance related issues. We are currently making use of DACs in various practical examples, which are far more complex than the simple chat example given in this paper. The objective of investing in several applications is to end up with a stable framework, that would for instance extend JGL. The issue of translating operations known from conventional collections to an asynchronous distributed context is however not entirely completed, and certain parts of the API might be affected by future modifications. We also explore approaches to express *content-based* publish/subscribe and specific algorithms to realize efficient matching. This is especially challenging in a mobile environment, where nodes might be disconnected, and objects might migrate from a node to another [17].

References

1. M. Altherr and M. Erzberger and S. Maffeis. iBus - A Software Bus Middleware for the Java Platform. In International Workshop on Reliable Middleware Systems, pages 43-53, October 1999.
2. K. Arnold and B. O'Sullivan and R.W. Scheifler and J. Waldo and J. Wollrath. The Jini Specification. Addison Wesley, 1999.
3. K.P. Birman. The Process Group Approach to Reliable Distributed Computing. In Communications of the ACM, 36:12, pages 36-53, December 1993.
4. J.-P. Briot and R. Guerraoui and K.-P. Löhr. Concurrency and Distribution in Object-Oriented Programming. In ACM Computing Surveys, September 1998.
5. D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. In Communications of the ACM, vol. 36, pages 90-102, September 1993.
6. M. Colan. InfoBus 1.2 Specificationa. Sun Microsystems Inc., February 1999.
7. Talarian Corporation. Everything You need to know about Middleware: Mission-Critical Interprocess Communication (White Paper). http://www.talarian.com/, 1999.
8. E. Freeman and S. Hupfer and K. Arnold. JavaSpaces Principles, Patterns, and Practice. Addison Wesley, 1999.
9. E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
10. D. Gelernter. Generative Communication in Linda. In ACM Transactions on Programming Languages and Systems, 7:1, pages 80-112, Januaray 1985.
11. R. Guerraoui. What object-oriented distributed programming does not have to be, and what it may be. In Informatik, 2, April 1999.
12. M. Happner and R. Burridge and R. Sharma. Java Message Service. Sun Microsystems Inc., October 1998.
13. M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. In ESEC/FSE 99 - Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7), September 1999.
14. IBM. Smalltalk Tutorial. http://www.smalltalksystems.com/references.htm/, 1995.
15. Sun Microsystems Inc. The Java Platform 1.2 API Specification. http://java.sun.com/products/jdk/1.2/, 1999.
16. Sun Microsystems Inc. The Java Collections Framework. http://java.sun.com/products/jdk/1.2/, 1999.

17. E. Jul and H. Levy and N. Hutchinson and A. Black. Fine-grained mobility in the Emerald System. In ACM Transactions on Computer Systems, 6:1, pages 109-133, February 1988.
18. P. Koenig,. Messages vs. Objects for Application Integration. In Distributed Computing, 2:3, pages 44-45, April 1999, BCI.
19. D. Lea Design for open systems in Java. In Second International Conference on Coordination Models and Languages, 1997. http://gee.cs.oswego.edu/dl/coord/.
20. D. Lea. Overview of package util.concurrent Release 1.2.5. http://gee.cs.oswego.edu/dl/classes/, October 1999.
21. ObjectSpace. JGL - Generic Collection Library. http://www.objectspace.com/products/jgl/, 1999.
22. B. Oki and M. Pfluegl and A. Siegel and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In Fourteenth ACM Symposium on Operating System Principles, pages 58-68, December 1993.
23. OMG. The Common Object Request Broker: Architecture and Specification. February 1998.
24. OMG. CORBAservices: Common Object Services Specification. December 1998.
25. Microsoft Co. DCOM Technical Overview (White Paper), 1999.
26. D. Powell. Group Communications. In Communications of the ACM, 39:4, pages 50-97, April 1996.
27. D. Rosenblum and A. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In Sixth European Software Engineering Conference/ACM SIGSOFT Fifth Symposium on the Foundations of Software Engineering, September 1997.
28. D. Schmidt and S. Vinoski. Overcoming Drawbacks in the OMG Event Service. In SIGS C++ Report magazine, 10, June 1997.
29. D. Skeen. Vitria's Publish-Subscribe Architecture: Publish-Subscribe Overview. http://www.vitria.com, 1998.
30. A. Stepanov and M. Lee. The Standard Template Library. Silicon Graphics Inc., October 1995.
31. Sun Microsystems Inc. Java Remote Method Invocation - Distributed Computing for Java (White Paper). http://java.sun.com/marketing/collateral/javarmi.html/, 1999.
32. TIBCO Inc. TIB/Rendezvous White Paper. http://www.rv.tibco.com/whitepaper.html, 1999.
33. J. Waldo and G. Wyant and A. Wollrath and S. Kendall. A Note on Distributed Computing. Sun Microsystems Inc., November 1994.
34. J. Waldo and G. Wyant and A. Wollrath and S. Kendall. Events in an RPC Based Distributed System. Sun Microsystems Laboratories Inc., November 1995.
35. A. Yonezawa and E. Shibayama and T. Takada and Y. Honda. Object-Oriented Concurrent Programming. In Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, pages 55-89, MIT Press, 1987.

Design Templates for Collective Behavior

Pertti Kellomäki[1] and Tommi Mikkonen[2]

[1] Software Systems Laboratory, Tampere University of Technology,
P.O Box 553, FIN-33101 Tampere, Finland
pk@cs.tut.fi

[2] Nokia Mobile Phones, Tieteenkatu 1, FIN-33720 Tampere, Finland
Tommi.Mikkonen@nokia.com

Abstract. While sequential behavior of single objects is fairly well understood, orchestrating the collective behavior emerging from the behaviors of individual objects continues to be a challenging task. This is especially true for distributed reactive systems.

The joint action paradigm is a design methodology that concentrates on the collective behavior of objects. Aspects of collective behavior are gradually introduced in a controlled manner in a specification. This paper presents how such aspects can be archived as generic templates, and instantiated in such a way that formal properties verified for a template become properties of its application. Both design and verification effort are reused when a template is applied.

1 Introduction

The traditional unit of modularity in software construction is a component, which is an open system with an interface and some internal state. A complete system is then composed of components. This approach works well at the implementation level, where the goal is to determine the distribution of code and data on the physical components of the system, and to assign responsibilities to teams and individual programmers. However, it is becoming evident that problems at the specification level call for something that extends across objects and ties them together [7].

The work reported here deals with the specification of *reactive distributed systems*, where the coordination of interactions between objects is of interest. Because of the inherent parallelism of such systems and the resulting complexity of the state space, it is very difficult to ascertain the correctness of a given design. An alternative to component based decomposition is to give functionality in an aspect-oriented fashion [15]. Each aspect specifies a view on the collective behavior of the system, and the complete specification is composed by superimposing the aspects. This enables a specifier to concentrate on the correctness of one aspect at a time.

The aspects often follow common idioms so it is advantageous to be able to archive and reuse them. The work presented here describes how this can be done using DisCo [1,10,14] specification language. We focus on safety properties,

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 277–295, 2000.
© Springer-Verlag Berlin Heidelberg 2000

specifically invariants. The main contribution of this paper is a method by which a design step introducing an aspect of collective behavior can be archived and applied in such a way that safety properties are transferred to the application.

A design step consists of a description of the context in which the step is applicable, and a description of the additional structure (variables and operations on them) that is superimposed on the context to solve a particular design problem. Design steps are expressed as DisCo specifications, so their safety properties may be verified as for any DisCo specification [12]. Applying a design step incurs proof obligations, but the proof obligations are typically considerably easier to verify than the safety properties obtainable from the design step.

The rest of the paper is organized as follows. Section 2 presents necessary background on action systems and the DisCo language. The main contribution of the paper is in Section 3, which describes how design steps are archived and applied. The formal basis of why safety properties are preserved is described in Section 4, and an extended example is given in Section 5. Related work is reviewed in Section 6, and conclusions are drawn in Section 7.

2 The DisCo Language and Action Systems

In this section we describe the joint action [2,3] approach to specification, the use of superposition, and the specification language DisCo.

2.1 Joint Actions

The formal basis of action systems lies in temporal logic. The specific temporal logic used here is Lamport's Temporal Logic of Actions [16] (TLA). An action system consists of a predicate on the initial state, and a number of state transitions called *actions*. An action is a TLA formula that describes a step in a computation. It gives the conditions under which the step may be taken, and defines the values of state variables after the step.

A *joint action* is a formula for a multi-party operation that describes a synchronization of the *participants* of the action. The states of the objects that do not participate the execution remain unchanged. An action may have *parameters*, which are similar to participants except that they denote values, not objects. The values assigned to the variables of the participants in an execution of an action may freely refer to the attributes of the other participants and the parameters of the action. A joint action defines inter-object cooperation at a high level of abstraction, where the focus is lifted from communication details and the behavior of individual objects to the collective behavior.

2.2 Superposition

Stepwise refinement in the DisCo approach is based on *superposition*. In superposition, new state variables are incrementally added to a specification until the desired level of detail is reached. The actions of the specification being refined may be augmented with assignments to the new state variables, but new assignments to state variables introduced earlier may not be given. The initial condition and the guards of the actions may also be strengthened.

Superposition preserves safety properties by construction, since all assignments to a state variable are given when the state variable is introduced. DisCo specifications are thus *closed world* specifications.

A specification may contain explicit nondeterminism in the form of action parameters. By augmenting actions with new constraints on the values the parameters may assume, one can constrain the nondeterminism at a later stage.

2.3 The DisCo Language

The experimental specification language DisCo (Distributed Cooperation) has been designed for the incremental development of specifications using superposition. The language syntactically enforces that only superposition steps are used, and it offers a convenient notation for expressing superposition.

The syntactic unit describing a superposition step in DisCo is a *layer*. A layer may import a number of specifications, whose state spaces and actions are merged. The layer may then refine the imported entities and introduce new ones.

State variables reside in *objects*, which are instances of *classes*. Objects are anonymous in the sense that it is not possible to refer to an object by name in a specification. Universal and existential quantification is used for referring to objects. An action lists the participants and their classes, the guard of the action, and the body consisting of assignments to state variables.

Figure 1 depicts a simple DisCo specification *L1*. It describes a world in which two objects of class C may arbitrarily swap the values of their i attributes. The same figure presents a superposition step *L2* that constrains the *swap* action in such a way that the result is a distributed exchange sort. Objects are also augmented with a counter that counts how many times they have been engaged in a *swap* action. The *extend* clause adds new variables to a class, and the ellipsis "..." in the action refinement denotes parts from the previous layer.

2.4 Tool Support

DisCo specifications have a straightforward operational interpretation, so even very high level specifications can be executed. To enable early validation of specifications, an animation tool [23] has been implemented.

Besides validation, the animation tool can be used for rudimentary state space exploration. More formal verification can be carried out using the PVS [21,20] theorem prover. DisCo specifications can be mapped to PVS theories [12] in order to verify invariant properties.

The current DisCo tools do not support parametric specifications as described in this paper, but we plan to extend them so that parametric specifications could be animated and verified in the same fashion as regular DisCo specifications.

```
layer L1 is
  class C is
   i : integer;
  end;

  action swap(c1, c2 : C) is
  when true do
   c1.i := c2.i;
   c2.i := c1.i
  end;
end;
```

```
layer L2 import L1 is
  extend C by
   left : C;
   swaps : integer := 0;
  end;

  refined swap(. . . ) is
  when . . . c2.left = c1
      and c1.i > c2.i do
   . . .
   c1.swaps := c1.swaps + 1;
   c2.swaps := c2.swaps + 1;
  end;
end;
```

Fig. 1. A DisCo superposition step

3 Archived Steps

In this section we describe the main contribution of the paper, namely how a superposition step can be archived and applied in such a way that design and verification effort is reused.

An archived step consists of a context layer describing the contextual constraints under which the step is applicable, and a solution layer describing the superposition step that can be taken if the constraints are satisfied. Contextual constraints are given in the form of a pattern of interactions between objects that needs to be present in a specification to which the step is to be applied.

If the pattern of objects and their interactions is present in a specification under construction, then an instance of the solution layer can be superimposed on it. The names of variables in the specification under construction may be different from those in the archived step, so the variables need to be linked together. This is accomplished by making the archived step parametric, and instantiating the parameters with entities from the specification under construction when the step is applied.

We use a simple example of token passing in the following. The example illustrates how collective behavior can be captured in an archived step. Section 5 gives a more extended example of archived steps and illustrates other aspects of the mechanism.

3.1 The Context Layer

The role of the context layer is to specify the behavior assumed when the archived step is applied, described operationally using joint actions. The formal parameters of the context layer denote entities (classes, actions, etc.) that need to be provided when the step is applied. The formal parameters are bound to entities in the specification to which the step is being applied.

In DisCo, parameterization is denoted by enclosing the parameters within square brackets. The formal parameter list of a layer must include all the entities (types, classes, functions, relations, initial conditions, assumptions, assertions and actions) defined in the layer. Each of the class parameters includes a list of variables the class should contain, and each of the action parameters includes a list of participants. The lists must agree with the definitions of the classes and actions in the layer.

A simple example of a context layer is presented in Figure 2. The specification describes a system in which objects perform *Act* operations nondeterministically. In order for a specification to match *TokenContext*, it needs to contain a class corresponding to *Object* and an action corresponding to *Act*.

```
layer TokenContext[
      Object :  class;
      Act(o) :  action]  is

  class Object  is
  end;

  action Act(o : Object)  is
  when true  do
  end;
end;
```

Fig. 2. A context layer

A context layer may not include initial conditions or assertions. This restriction is not essential, but it makes it easier to establish refinement between the context layer of a step and a specification to which the step is being applied. If initial conditions are needed for verification of safety properties, they can be introduced in the solution layer.

3.2 The Solution Layer

The solution layer refines the entities in the context layer and introduces new entities, including initial conditions and assertions. The formal parameter list of the solution layer consists of the new entities in the layer. Since they are to be introduced as new entities in the application, the formal parameters are not bound to entities in the specification under construction. Instead, fresh names are provided for them.

An example of a solution layer that can be superimposed on *TokenContext* is presented in Figure 3. It describes how objects coordinate their *Act* operations by circulating a token that grants permission to perform the operation.
The solution layer introduces a new class *Token*, an initial condition *SingleToken* constraining the number of tokens, and a new action to pass the token around.

```
layer TokenSolution[
 Token(at) : class; SingleToken : initial condition;
 SingleEnabled : assertion; PassToken(t,o) : action;
 Act(o,t) : action] import TokenContext is

  class Token is
   at : Object;
  end;

  initially SingleToken is ∀ t1, t2 : Token :: t1 = t2;

  assert SingleEnabled is
   not exists o1, o2 : Object ::
        o1 /= o2 and enabled(Act(o=o1)) and enabled(Act(o=o2));

  action PassToken(t : Token; o : Object) is
  when true do
   t.at := o;
  end;

  refined Act(... t : Token) of Act(...) is
  when ... t.at = o do
   ...
  end;
 end;
```

Fig. 3. A solution layer

The formal safety property guaranteed by the solution layer is expressed as assertion *SingleEnabled*, where *enabled(Act(o=o1))* denotes $\exists tt : Token ::$ $(Guard(Act))(o1, tt)$, i.e., such a token exists that the guard of *Act* is true. The assertion is easy to verify for the solution layer.

3.3 Applying a Step

Applying an archived step to a specification S under construction involves providing actual parameters for the formal parameters of the layers. This links entities in S with entities in the archived steps.

The actual parameters for the formal parameters of the context layer are entities drawn from S. For the data related entities (classes, types, relations), there needs to be a one to one correspondence. Each action in the context layer may correspond to zero or more actions in S, so there is a list of actions corresponding to each formal action. For a formal action that is refined by the solution layer, the list should contain exactly one actual action. The actual action to be refined is thus always uniquely determined.

It is simple to check that the actual parameter provided for each data-related formal parameter agrees with the declaration of the formal entity in the context layer. For each action provided as an actual parameter, a proof obligation of refinement is obtained.

Consider the following specification:

```
layer Uncoordinated is
  class Node is
   v : integer;
  end;

  class Medium(1) is
   v : integer;
  end;

  action Send(sender, receiver : Node; m : Medium) is
  when true do
   m.v := sender.v;
   receiver.v := sender.v;
  end;
end;
```

The specification describes a system in which a sender and a receiver can communicate synchronously over a medium shared by all nodes (the parenthesized one indicates that there is exactly one medium). The token passing strategy can be applied to coordinate the use of the medium. Figure 4 presents an instance of the solution layer applicable to *Uncoordinated*.

```
layer Coordinated import Uncoordinated is
  assert
   refines(Uncoordinated,
          TokenContext[Uncoordinated.Node; {Uncoordinated.Send(sender)}])
  instance TokenSolution[Token(at); SingleToken;
                               SingleEnabled; PassToken(t,n); Send(n,t)]
  class Token is
   at : Object;
  end;

  initially SingleToken is ∀ t1, t2 : Token :: t1 = t2;

  invariant SingleEnabled is
   not ∃ o1, o2 : Object :: o1 /= o2
         and enabled(Send(sender=o1))
         and enabled(Send(sender=o2));

  action PassToken(t : Token; n : Node) is
  when true do t.at := n; end;

  refined Send(... t : Token) of Send(...) is
  when ... t.at = n do ... end;
 end;
```

Fig. 4. Token passing applied to specification *Uncoordinated*

The *assert refines* clause expresses a proof obligation to show that *Uncoordinated* is a refinement of the given instance of *TokenContext*. There are no requirements for the actual parameter corresponding to class *Object* in the context layer, since the formal class does not have any internal structure. The extended example in Section 5 presents a more complex binding of an actual class to a formal class.

The proof obligation for each formal action–actual action pair is to establish that the guard and body of the actual action imply the guard and body of the instance of the formal. The instance of the formal action is derived by replacing the formal relations, types and classes have been replaced by their actual counterparts, and replacing the names of the participants and parameters of the formal action by the names provided in the binding. If the list of actions provided for a formal action is empty, no proof obligation is obtained (one can always think that a corresponding action exists, but its guard is identically false). Furthermore, the actual actions not paired with formal actions must be stuttering with respect to the variables provided as actual parameters.

The instance of the formal action *act* is

```
action Send(sender : Node) is
when true do
end;
```

and the corresponding proof obligation is

$$(true \wedge m.v' = sender.v \wedge receiver.v' = sender.v) \Rightarrow true, \tag{1}$$

which is trivial to verify (primed variables denote the values of the variables after the execution). Section 5 presents examples of more complicated proof obligations.

Intended invariants of an archived step are given as assertions in the solution layer of the step. The solution layer may also include assumptions that can be used when verifying the assertions. When a solution layer is instantiated, assertions in the solution layer become invariants in its instance, and assumptions become assertions (proof obligations). If an assertion has been verified in the archived step, it is sufficient to verify the refinement discussed above and to prove that the proof obligations hold, in order to establish that the assertion is an invariant of the application of the step. Verifying (1) thus establishes *SingleEnabled* in *Coordinated*.

3.4 Tool Support

The instance in Figure 4 was produced manually, but our intent is to provide a mechanical tool to perform the instantiation of archived steps. Such a tool should make matching a specification under construction with a context layer easy. Much of the matching process could be automated using a unification style approach.

The instance of the solution layer is linked to the archived step by the *refines* and *instance* clauses. This is useful for documentation purposes, since it tells the reader of the specification how the layer was obtained. It can also be used for checking that the instance agrees with the archived step, in order to catch modifications to either the instance or the archived step that would invalidate the invariants verified for the step.

4 Formal Basis

In this section we explain the formal basis of why safety properties verified for an archived step hold when it is applied.

Informally the argumentation proceeds as follows. If we can establish that a specification S to which we wish to apply a step implies an instance of the context of the step, then the conjuncts of the actions of the context are in a sense contained in the conjuncts of the actions of S. Applying the solution layer adds equal conjuncts to both S and the solution layer. Hence anything implied by the specification obtained by superimposing the solution layer on the context is also implied by the specification obtained by superimposing the solution layer on S.

More formally, a superposition step is a function $\mathcal{U}$ that maps specifications to specifications. Consider a design step consisting of a context part L_0 and a superposition step $\mathcal{U}$. Verifying a safety property P for $\mathcal{U}(L_0)$ means verifying

$$\mathcal{U}(L_0) \Rightarrow P. \tag{2}$$

When the step is applied, we create an instance $\widehat{L_0}$ of the context and an instance $\widehat{\mathcal{U}}$ of the superposition step by replacing the formal relations, types and classes with the actual entities provided. Since there is a trivial refinement mapping between $\mathcal{U}(L_0)$ and $\widehat{\mathcal{U}}(\widehat{L_0})$, it is true that that

$$\widehat{\mathcal{U}}(\widehat{L_0}) \Rightarrow \widehat{P}, \tag{3}$$

where $\widehat{P}$ denotes an instance of P.

We want to derive a sufficient condition C so that the safety property P also holds in an application $\widehat{\mathcal{U}}(S)$ of the step to specification S, i.e.

$$C \Rightarrow ((\mathcal{U}(L_0) \Rightarrow P) \Rightarrow (\widehat{\mathcal{U}}(S) \Rightarrow \widehat{P})). \tag{4}$$

We proceed by examining actions $\mathcal{A}_{\widehat{\mathcal{U}}(S)}$ in $\widehat{\mathcal{U}}(S)$ and $\mathcal{A}_{\widehat{\mathcal{U}}(\widehat{L}_0)}$ in $\widehat{\mathcal{U}}(\widehat{L}_0)$. Each action $\mathcal{A}_{\widehat{\mathcal{U}}(S)}$ is a refinement of some action $\mathcal{A}_S$ in S, which is further bound to an action $\mathcal{A}_{\widehat{L}_0}$ in $\widehat{L}_0$ by the formal parameter list of L_0. The superposition step $\widehat{\mathcal{U}}$ may refine $\mathcal{A}_S$ to a disjunction

$$\mathcal{A}^0_{\widehat{\mathcal{U}}(S)} \vee \cdots \vee \mathcal{A}^k_{\widehat{\mathcal{U}}(S)} \tag{5}$$

and $\mathcal{A}_{\widehat{L_0}}$ to

$$\mathcal{A}^0_{\widehat{\mathcal{U}}(\widehat{L_0})} \vee \cdots \vee \mathcal{A}^k_{\widehat{\mathcal{U}}(\widehat{L_0})}, \tag{6}$$

but there is a one to one correspondence between the disjuncts (a superposition step always introduces a fixed number of refinements). It is thus sufficient to consider a single pair of actions $\mathcal{A}_{\widehat{\mathcal{U}}(S)}$ and $\mathcal{A}_{\widehat{\mathcal{U}}(\widehat{L_0})}$.

Let $\mathcal{A}_{\widehat{L_0}}$ be defined as

$$\mathcal{A}_{\widehat{L_0}} = \exists p_1, \ldots, p_n : F_{\widehat{L_0}}(p_1, \ldots, p_n). \tag{7}$$

Since a superposition step cannot remove conjuncts from an action, its refinement is of the form

$$\begin{aligned}\mathcal{A}_{\widehat{\mathcal{U}}(\widehat{L_0})} = {} & \exists p_1, \ldots, p_n, q_1, \ldots, q_m : \\ & F_{\widehat{L_0}}(p_1, \ldots, p_n) \\ & \wedge F^{new}_{\widehat{\mathcal{U}}}(p_1, \ldots, q_m).\end{aligned} \tag{8}$$

Similarly, let $\mathcal{A}_S$ be defined as

$$\mathcal{A}_S = \exists p_1, \ldots, p_l : F_S(p_1, \ldots, p_l). \tag{9}$$

Applying $\widehat{\mathcal{U}}$ to it adds the same participants and parameters $q_1, \ldots, q_m$, and conjuncts $F^{new}_{\widehat{\mathcal{U}}}$:

$$\begin{aligned}\mathcal{A}_{\widehat{\mathcal{U}}(S)} = {} & \exists p_1, \ldots, p_l, q_1, \ldots, q_m : \\ & F_S(p_1, \ldots, p_l) \\ & \wedge F^{new}_{\widehat{\mathcal{U}}}(p_1, \ldots, q_m)\end{aligned} \tag{10}$$

If we assume

$$F_S(p_1, \ldots, p_l) \Rightarrow F_{\widehat{L_0}}(p_1, \ldots, p_n), \tag{11}$$

it is possible to write $\mathcal{A}_S$ equivalently as

$$\begin{aligned}\mathcal{A}_{\widehat{\mathcal{U}}(S)} = {} & \exists p_1, \ldots, p_l, q_1, \ldots, q_m : \\ & F_{\widehat{L_0}}(p_1, \ldots, p_n) \\ & \wedge F_S(p_1, \ldots, p_l) \\ & \wedge F^{new}_{\widehat{\mathcal{U}}}(p_1, \ldots, q_m)\end{aligned} \tag{12}$$

This is a simple consequence of the tautology $(Q \Rightarrow R) \Rightarrow (Q \Leftrightarrow Q \wedge R)$. From (12), and (8) it follows that

$$\mathcal{A}_{\widehat{\mathcal{U}}(S)} \Rightarrow \mathcal{A}_{\widehat{\mathcal{U}}(\widehat{L_0}))}. \tag{13}$$

The reasoning above applies to all actions in $\widehat{\mathcal{U}}(S)$, since each of them is a refinement of some action in S and thus a refinement of some action in $\widehat{L}_0$. It follows that if $\widehat{P}$ is a safety property of $\widehat{\mathcal{U}}(\widehat{L}_0)$, no action in $\widehat{\mathcal{U}}(S)$ can invalidate it:

$$(\widehat{\mathcal{U}}(\widehat{L}_0) \Rightarrow \widehat{P}) \Rightarrow (\widehat{P} \wedge \mathcal{A}_{\widehat{\mathcal{U}}(S)} \Rightarrow \widehat{P}') \tag{14}$$

L_0 does not contain any initial conditions, so the initial conditions of $\widehat{\mathcal{U}}(\widehat{L_0})$ are by construction present in $\widehat{\mathcal{U}}(S)$ as well, as they are introduced by $\widehat{\mathcal{U}}$. We thus conclude that (11) is a sufficient condition for the preservation of safety properties of $\mathcal{U}(L_0)$ in $\widehat{\mathcal{U}}(S)$.

5 An Extended Example

In this section we give an extended example of an archived step that illustrates some aspects not present in the token passing example. The example deals with the common problem of communicating some data over a channel that cannot directly represent the data. At the sending end a datum to send is decomposed into lower level data, and at the receiving end it is composed back to a higher level datum. The obvious safety property is that the lower level channel should implement a higher level channel.

Figure 5 presents a parameterized DisCo layer describing the contextual constraints of the decomposition–composition step. This time the context is somewhat more complicated, involving variables and state changes. The specification describes a system in which data is produced to the sending buffer of the sender, sent to the receiver, and consumed from the receiving buffer of the receiver.

Besides the entities introduced in *DecomposeComposeContext*, its formal parameters include the type *lowT* and the functions *decompose* and *compose*, which are not used in the layer. The reason for their appearance is that they are entities used in the solution layer that should nevertheless be provided when the step is applied.

Figure 6 depicts the solution part of the step. In the solution, a datum to be sent is first decomposed into lower level date, and the lower level data are sent one by one to the receiver. Transmitting the final low level datum corresponds to the high level send action,

The invariant ensured by the solution is *lowBufferComposition*, which states that the composition of the low level data in the *lowBuffer* variables equals the first item to be sent in the sender's high level sending queue. In particular, this means that a receiver does not need access to the sender's high level sending queue in action *sendLast*. It is sufficient to access the local low level receiving queue. The specification does not specify the mechanism by which the receiver recognizes the last low level datum, since there are several possibilities a particular application could use.

The invariant can be verified using the initial conditions *SingleSender* and *SingleReceiver* which constrain the number of objects, and the assumption *IsInverse* which describes the properties required from the functions.

Figure 7 presents a specification to which we wish to apply the decomposition – composition step. It describes a weather sensor that measures temperature and humidity and sends the measurements to a recorder that stores the sequence of measurements. Actions *senseTemperature* and *senseHumidity* model the independent sensing of the environment, and action *measure* models reading the sensors and storing the measurement to be transmitted.

```
layer DecomposeComposeContext[
 Sender(dataToSend), Receiver(receivedData) : class;
 highT, lowT : type; decompose, compose : function;
 produce(s,d), send(s,r), consume(r) : action] is

  class Sender is
   dataToSend : sequence hightT;
  end;

  class Receiver is
   receivedData : sequence hightT;
  end;

  action produce(s : Sender; d : highT) is
  when true do
   s.dataToSend := s.dataToSend & <d>;
  end;

  action send(s : Sender; r : Receiver) is
  when size(s.dataToSend) > 0 do
   r.receivedData := r.receivedData & <head(s.dataToSend)>;
   s.dataToSend := tail(s.dataToSend);
  end;

  action consume(r : Receiver) is
  when size(r.receivedData) > 0 do
   r.receivedData := tail(r.receivedData);
  end;
end;
```

Fig. 5. Context part of decomposition–composition

The specification does not include functions corresponding to *compose* and *decompose*, so they are introduced in a separate layer *WeatherFunctions* (Figure 8) before applying the step. Action *measure* is also augmented with a parameter matching *d* in *produce*.

Applying the step to *WeatherFunctions* yields an instance of *Decompose-ComposeSolution*. Figure 9 presents an excerpt of the instance. Actions have been omitted, as they are simply derived by making the substitutions of actual parameters to formal parameters.

Compared to the token passing example, an application of the decompose–compose step incurs more complicated proof obligations. The proof obligations related to action *produce* in the archived step is

```
layer DecomposeComposeSolution[
 Sender(lowBuffer), Receiver(lowBuffer) : class;
 emptyBuffers, SingleSender, SingleReceiver : initial condition;
 IsInverse : assumption; lowBufferComposition : assertion;
 prepareSend(s),lowSend(s,r),sendLast(s,r) : action]
import DecomposeComposeContext is
  extend Sender by lowBuffer : sequence lowT; end;

  extend Receiver by lowBuffer : sequence lowT; end;

  initially emptyBuffers is
   ∀ s : Sender; r : Receiver :: s.lowBuffer = < > and r.lowBuffer = < >;

  initially SingleSender is ∀ s1, s2 : Sender :: s1 = s2;
  initially SingleReceiver is ∀ r1, r2 : Receiver :: r1 = r2;

  assumption IsInverse is
    ∀ ht : highT :: compose(decompose(ht)) = ht;

  assert lowBufferComposition is
   ∀ s : Sender; r : Receiver ::
     size(r.lowBuffer & s.lowBuffer) /= 0
     => compose(r.lowBuffer & s.lowBuffer) = head(s.dataToSend);

  action prepareSend(s : Sender) is
  when size(s.lowBuffer) = 0 and size(s.dataToSend) > 0 do
   s.lowBuffer := decompose(head(s.dataToSend));
  end;

  action lowSend(s : Sender; r : receiver) is
  when size(s.lowBuffer) > 1 do
   r.lowBuffer := r.lowBuffer & <head(s.lowBuffer)>;
   s.lowBuffer := tail(s.lowBuffer);
  end;

  refined sendLast(...) of send(...) is
  when ... size(s.lowBuffer) = 1 do
   ...
   s.lowBuffer := < >;
   r.lowBuffer := < >;
  end;
end;
```

Fig. 6. Solution part of decomposition–composition

$$
\begin{aligned}
&\quad (size(s.sendingBuffer) = 0 \\
&\quad \wedge m = Measurement(s.temperatureReading, s.humidityReading) \\
&\quad \wedge s.sendingBuffer' = \\
&\qquad\qquad \langle Measurement(s.temperatureReading, s.humidityReading)\rangle) \\
&\Rightarrow \\
&\quad (true \wedge s.sendingBuffer' = s.sendingBuffer \,\&\, \langle m\rangle),
\end{aligned} \tag{15}
$$

```
layer WeatherSensor is
  type Measurement is
   temperature : real;
   humidity : real;
  end;

  class Sensor is
   temperatureReading : real;
   humidityReading : real;
   sendingBuffer : sequence Measurement;
  end;

  class Recorder is measurements : sequence Measurement; end;

  action senseTemperature(s : sensor; t : real) is
  when true do
   s.temperatureReading := t;
  end;

  action senseHumidity(s : sensor; h : real) is
  when true do
   s.humidityReadon := h;
  end;

  action measure(s : Sensor) is
  when size(s.sendingBuffer) = 0 do
   s.sendingBuffer := <Measurement(s.temperatureReading,
                                              s.humidityReading)>;
  end;

  action transmit(s : Sensor; r : receiver) is
  when size(s.sendingBuffer) = 1 do
   r.measurements := r.measurements & <head(s.sendingBuffer)>;
   s.sendingBuffer := < >;
  end;
end;
```

Fig. 7. Specification of a weather station

which is relatively straightforward to verify. The proof obligation related to action *send* is

$$\begin{aligned}
&\quad (size(s.sendingBuffer) = 1\\
&\quad \wedge r.measurements' = r.measurements \,\&\, \langle head(s.sendingBuffer)\rangle\\
&\quad \wedge s.sendingBuffer' = \langle\rangle)\\
&\Rightarrow\\
&\quad (size(s.sendingBuffer) > 0\\
&\quad \wedge r.measurements' = r.measurements \,\&\, \langle head(s.sendingBuffer)\rangle\\
&\quad \wedge s.sendingBuffer' = tail(s.sendingBuffer)).
\end{aligned} \tag{16}$$

```
layer WeatherFunctions  import WeatherSensor  is
  function MeasurementToReals(m : Measurement) : sequence real  is
   <m.temperature, m.humidity>
  end;

  function RealsToMeasurement(s : sequence real) : Measurement  is
   Measurement(head(s), head(tail(s)))
  end;

  refined measure(... m : Measurement)  of measure(...)  is
  when ... m = Measurement(s.temperatureReading, s.humidityReading)
  do ...  end;
end;
```

Fig. 8. Adapting *WeatherSensor* to *DecomposeComposeContext*

This is also easy to verify. As there is no action in *WeatherSensor* corresponding to *consume*, no proof obligations arise.

The assumption *IsInverse* in the solution layer gives rise to a further proof obligation as assertion *IsInverse* in *WeatherSolution*. Verifying it is also a simple exercise in reasoning about records and sequences.

6 Related Work

Formal refinement of specifications into executable programs has been extensively studied (see e.g. [4,19]). Our work is different from the mainstream refinement research because of the use of superposition. A superposition step is a refinement step by construction, so refinement need not be verified. Instead, we are interested in safety properties related to the new variables introduced in a superposition step. These properties may link newly introduced variables to the higher level variables (data refinement), or they may be e.g. consistency properties of the new variables.

Compared to refinement of specifications into sequential programs, our work has somewhat different focus. Instead of algorithmic complexity, we are interested in managing the complexity emerging from interactions of objects. In this context, complex local computations can often be abstracted away using nondeterminism, and sophisticated data refinement is not necessary. For a fully formal derivation of distributed systems the sequential computations embodied in actions would also need to be derived, but this is not our present goal.

Archiving design steps is well established in the context of sequential programs. The Specware [11] system and others developed in the Kestrel Institute have been notably successful in the derivation of executable programs from algebraic specifications [22,5]. In Specware, refinement is based on *interpretations.* An interpretation $\mathcal{A} \to S$ is a pair of *morphisms* $S \to S'$ and $\mathcal{A} \to S'$, where $\mathcal{S}'$ is a definitional extension of S. In our construction $S \to S'$ is the superposition relation, and $\mathcal{A} \to S'$ is the composition of instantiation and refinement.

```
layer WeatherSolution  import WeatherFunctions
assert
  refines(WeatherFunctions,
        DecomposeComposeContext[WeatherFunctions.Sensor(sendingBuffer),
               WeatherFunctions.Recorder(measurements),
               WeatherFunctions.Measurement, real,
               WeatherFunctions.MeasurementToReals,
               WeatherFunctions.RealsToMeasurement,
               {WeatherFunctions.measure(s,d)},
               {WeatherFunctions.transmit(s,r)}, {}]
instance DecomposeComposeSolution[emptyBuffers, SingleSensor,
            SingleRecorder, IsInverse, lowBufferComposition, Sensor(lowBuffer),
               Recorder(lowBuffer), prepareSend(s), lowSend(s,r), sendLast(s,r)]
is
  extend Sensor  by lowBuffer : sequence real;  end;
  extend Recorder  by lowBuffer : sequence real;  end;

  initially emptyBuffers  is
   ∀ s : Sensor; r : Recorder :: s.lowBuffer = < > and r.lowBuffer = < >;

  initially SingleSensor  is ∀ s1, s2 : Sensor :: s1 = s2;
  initially SingleRecorder  is ∀ r1, r2 : Recorder :: r1 = r2;

  assert IsInverse  is
   ∀ ht : Measurement ::
        RealsToMeasurement(MeasurementToReals(ht)) = ht;

  invariant lowBufferComposition  is
   ∀ s : Sensor; r : Recorder ::
     r.lowBuffer & s.lowBuffer /= < > =>
    RealsToMeasurement(r.lowBuffer & s.lowBuffer) = head(s.sendingBuffer);

              ⋮
end;
```

Fig. 9. Application of *DecomposeComposeSolution* to *WeatherFunctions*

By restricting to the special case, we manage without the rather heavy mathematical machinery underneath Specware. Avoiding mathematics is of course not a value in itself, but our approach meshes well with how development is normally done in DisCo. It also uses concepts that are close to those of programming, which has been a design goal when developing the DisCo approach.

Our work is also connected with *patterns* [8,6]. A specification in the closed world approach embodies a solution and the context in which the solution is applicable. Together with an informal problem statement, these meet a common definition of what constitutes a pattern. A joint action specification formally describes relations and interactions between objects, which in pattern literature are usually given informally using class diagrams accompanied with prose and descriptions of message interchange. Behavioral patterns can easily be expressed as closed world joint action specifications. Work on formalizing patterns in the joint action formalism has been reported in [18].

Compositional TLA (cTLA) [9] is an approach closely related to DisCo, as both are specification languages based on Temporal Logic of Actions [16]. The languages support and encourage quite different styles of specification, though. Compared to DisCo, cTLA has a more component oriented flavor: a specification is composed of *processes*, which are subsystems described in TLA. Each process has a set of private state variables, which are only accessible to the actions of the process. When a new process is composed, actions of the component processes are synchronized, and action parameters are used for sharing values among the synchronized actions.

The cTLA *pattern integration* [17] operation can be used for merging the state spaces of component processes. Integration is similar to composition, but additionally one gives a new set of state variables. The state variables of the components are replaced with expressions involving the new state variables. The result is a process that has the properties of its components, but a flat rather than a hierarchical structure. The pattern integration operation can be used to achieve results similar to those of our approach, but with a considerably different style.

At a first glance, archived joint actions specifications may resemble e.g. Ada generic packages or C++ templates. There is an important difference brought about by the use of superposition, however. Instead of components, our archived units describe *aspects* of behavior. In the programming language world the closest analogy is *aspect oriented programming* [13], where fragments of code called aspects are woven into a complete program by an aspect weaver. The ellipsis "..." in DisCo plays the role of the aspect weaver.

7 Conclusions and Further Work

We have presented how aspects of collective behavior can be archived and applied as parametric joint action specifications. Applying such archived aspects facilitates the reuse of both design and verification effort. Safety properties may be verified for an archived specification. These properties are transferred to an application of the specification by discharging a set of proof obligations that are typically easier to verify than the safety property itself.

Further work is needed for finding aspects that are worthwile to archive and verify. In connection with the DisCo method we see the greatest potential in situations where atomicity needs to be ensured e.g. by some form of locking. Implementing data abstractions is another possible area of application. We plan to extend the current DisCo tools so that parametric specifications could be animated and verified.

Acknowledgments. This work has been partly funded by the Academy of Finland, project 757473.

References

1. The DisCo project WWW page. At URL http://disco.cs.tut.fi/ on the World Wide Web, 1999.
2. R. J. R. Back and R. Kurki-Suonio. Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.
3. R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with a centralized control. *Distributed Computing*, (3):73–87, 1989.
4. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.
5. Lee Blaine, Li-Mei Gilham, Junbo Liu, Douglas R. Smith, and Stephen Westfold. Planware – domain-specific synthesis of high-performance scheduler. In *Proceedings of the Thirteenth Automated Software Engineering Conference*, pages 270–280. IEEE Computer Society Press, 1998.
6. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns.* John Wiley & Sons, 1996.
7. J. O. Coplien. Idioms and patterns as architectural literature. *IEEE Software*, pages 36–42, January 1997.
8. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, Reading, MA, 1995.
9. P. Herrmann, G. Graw, and H. Krumm. Compositional specification and structured verification of hybrid systems in cTLA. In *Proceedings of the 1st IEEE International Symposium on Object-oriented Real-time distributed Computing - (ISORC'98)*, pages 335–340. IEEE Computer Society Press, 1998.
10. H.-M. Järvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systä. Object-oriented specification of reactive systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71. IEEE Computer Society Press, 1990.
11. R. Juellig, Y. Srinivas, and J. Liu. SPECWARE: An advanced environment for the formal development of complex software systems. Lecture Notes in Computer Science, Vol. 1101, Springer–Verlag, 1996.
12. Pertti Kellomäki. Verification of reactive systems using DisCo and PVS. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods.* Lecture Notes in Computer Science, Vol. 1313, pages 589–604. Springer–Verlag, 1997.
13. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference. Lecture Notes in Computer Science*, Vol. 1241, pages 220–242. Springer–Verlag, 1997
14. Reino Kurki-Suonio. Fundamentals of object-oriented specification and modeling of collective behaviors. In H. Kilov and W. Harvey, editors, *Object-Oriented Behavioral Specifications*, pages 101–120. Kluwer Academic Publishers, 1996.
15. Reino Kurki-Suonio and Tommi Mikkonen. Abstractions of distributed cooperation, their refinement and implementation. In B. Krämer, N. Uchihira, P. Croll, and S. Russo, editors, *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 94–102. IEEE Computer Society, April 1998.
16. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

17. Arnulf Mester and Heiko Krumm. Formal behavioural patterns for the tool-assisted design of distributed applications. In Hartmut König, Kurt Geihs, and Thomas Preuß, editors, *IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 97)*, pages 235–248. Chapman & Hall, 1997.
18. Tommi Mikkonen. Formalizing design patterns. In B. Werner, editor, *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pages 115–124. IEEE Computer Society, April 1998.
19. Carroll Morgan. *Programming from Specifications: Second Edition.* Prentice Hall International, Hempstead, UK, 1994.
20. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction.* Lecture Notes in Artificial Intelligence, Vol. 607, pages 748–752. Springer–Verlag, 1992.
21. Sam Owre, John Rushby, Natrajan Shankar, and Friedrich von Henke. Formal verification of fault-tolerant architecures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.
22. D. Smith and E. Parra. Transformational approach to transportation scheduling. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, pages 60–68. IEEE Computer Society Press, September 1993.
23. Kari Systä. A graphical tool for specification of reactive systems. In *Proceedings of the Euromicro'91 Workshop on Real-Time Systems*, pages 12–19, Paris, France, June 1991. IEEE Computer Society Press.

Ionic Types

Simon Dobson[1] and Brian Matthews[2]

[1] Department of Computer Science, Trinity College, Dublin IE
simon.dobson@cs.tcd.ie
[2] Information Technology Department, CLRC Rutherford Appleton Laboratory,
Oxfordshire UK
b.m.matthews@rl.ac.uk

Abstract. We are interested in the class of systems for which the satisfaction of code dependencies is a dynamic process rather than one which is determined purely at load-time. Examples include dynamic delegation, mobile code and agent systems. Such systems exhibit properties which are not well-captured by current typing models. We describe a system of *ionic* object types which capture these effects and allow them to be analysed within a standard object type framework. We show how ionic types improve the modeling of various forms of dynamic object completion including certain aspects of security checking, delegation and method update, contrast them with other approaches to the same problems, and sketch a possible ionic extension to Java.

1 Introduction

Programming languages exhibit a constant tension between static and dynamic checking. The former attempts to catch problems at compile-time, but inhibits many useful dynamic decisions; the latter provides more run-time flexibility at a cost of run-time type errors. This tension is exhibited by many programming approaches of current interest including dynamic adaptation, code mobility and component-based systems. Designers must balance the need for dynamic flexibility against a desire for static correctness checks.

Such systems are representative of a broad class in which the satisfaction of code dependencies is a dynamic process rather than one fixed at compile- or load-time. This dynamism manifests itself in various guises. In a delegation-based system the object which actually satisfies a method may be re-assigned as computation progreesses. For a mobile code system, it occurs when a piece of code migrates into a foreign environment and is tacitly re-linked with that environment's functions for display, file access *etc.* Under certain security regimes the environment may refuse to provide some expected functions, causing execution to fail at run-time. The important point is that the application depends on the availability of the functions and could *never* execute successfully under that security regime, so many of the dynamic security checks – although required in other, more complex cases – might better be performed statically. Intuitively it seems preferable for code whose requirements can never be accommodated to fail before execution.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 296–312, 2000.
© Springer-Verlag Berlin Heidelberg 2000

The unifying theme is that an object's response to a method can be changed as computation progresses. These changes are not generally arbitrary, but have a well-defined structure and require that some dependencies are maintained. If we capture these structures and dependencies uniformly within the type framework we can model their behaviours more accurately and obtain better predictions of an application's long-term behaviour.

We have been exploring the notion of *ionic types* which allow functionality to be deliberately extracted from an object while retaining the necessary dependencies. The resulting *ion* may then be resolved against another object implementing the extracted functions. Ionic types provide a direct and type-safe model for examining the dynamic satisfaction of code dependencies in its various guises, providing new perspectives on phenomena including inheritance, mobile agents, delegation and method update.

In this preliminary study we present a system of ionic types expressed within the object calculus of Abadi and Cardelli[2]. We introduce ions as variants on standard objects by providing types and operations for removing and resolving functionality, and show how the ideas apply to both objects and classes. We discuss some of the issues in static *versus* dynamic checking of ions, and then describe the use of ions in a number of scenarios involving dynamic changes in behaviour. We contrast our work with that of others and briefly sketch a possible embedding of ions into Java, before concluding with some directions for the future.

2 Necessary formalism

We present ions using the system $Ob_{1<:}$, one of the simplest members of the family of object calculi developed by Abadi and Cardelli – readers familiar with the system may want to skip this section. It should be noted that ions have no essential dependence on the features of this system and can easily be generalised to other calculi.

A,B ::=	types
X	type variable
K	ground type
Top	the biggest type
$[l_i v_i : A_i^{i \in 1..n}]$	object type
$A \rightarrow B$	function type

a,b ::=	terms
x	variable
$[l_i = \varsigma(self : A)a_i^{i \in 1..n}]$	object literal
$a.l$	member access
$a.l \Leftarrow \varsigma(x : A)b$	member update
$\lambda(x : A)b$	function literal
$b(a)$	function application

Fig. 1. Syntax of the $Ob_{1<:}$ calculus

The object calculus is formed using the types and terms from figure 1. An object type is represented by a sequence of labelled members, all labels distinct. An object literal maps labels to values, where a value may include a reference to the object itself. These self-referring members are the *methods* in the object, and are constructed using the ς (sigma) binder. A method term such as $\varsigma(\mathit{self} : O)b$ binds *self* within b to the object on which the method is called. More complex methods may be built using a λ-term for the body of the method. The "dot" operator is used to access the members of an object, and will implicitly bind *self* in any methods accessed. For example the object type $O \equiv [\mathit{val} : \mathit{Int}, \mathit{get} : \mathit{Unit} \rightarrow \mathit{Int}]$ defines the type of an object with two members, an integer *val* and a function *get* taking no arguments and returning an integer. One possible object with this type is $o \equiv [\mathit{val} = 1, \mathit{get} = \varsigma(\mathit{self} : O)\lambda()\mathit{self}.\mathit{val} + 1]$, where the *get* member is a method with self reference *self* and a body consisting of a function returning the value of *val* plus one.

Note that the use of methods does not affect the type signatures of members, since ς-binding is performed implicitly by the dot operator. This has a number of implications, most notably that, using the example above, *o.get* (with no application) generates an ordinary function whose free *self* references are bound to the object from which it was accessed. Note also that the usual interpretation of the calculus is purely functional, so assigning to a member generates a new object rather than an update-in-place.

(Sub Object)

$$\frac{\Gamma \vdash \upsilon_i B_i <: \upsilon_i' B_i'}{[l_i \upsilon_i : B_i^{i \in 1..n+m}] <: [l_i \upsilon_i' : B_i'^{i \in 1..n}]} \quad \forall i \in \{1..n\},\ \upsilon_i \in \{^0, ^+\}$$

(Sub invariant)

$$\frac{\Gamma \vdash B}{\Gamma \vdash {}^0B <: {}^0B}$$

(Sub covariant)

$$\frac{\Gamma \vdash B <: B'}{\upsilon B <: {}^+B'} \quad \upsilon \in \{^0, ^+\}$$

Fig. 2. $Ob_{1<:}$ object sub-typing

Object sub-typing is generally invariant: an object type A is a sub-type of another object type B iff A has at least the members of B with the same types. However, methods may be decorated with $^+$ to mark them as covariant[1]. The ordinary form of method declaration, with no annotation, is considered a shorthand for an invariance decoration ^{0}. (Figure 2, where the υ_i represent decorations.) For our work we use covariance solely for encoding classes.

Classes are not regarded as primitive but are encoded as objects: a class for an object type is simply an object which builds instances of that type with common definitions for the members, and there may be several classes generating objects of a given object type. For an object type $A \equiv [l_i : A_i^{i \in 1..n}]$ the corresponding class type $\mathit{Class}(A) \equiv [\mathit{new}^+ : A, l_i^+ : A \rightarrow A_i^{i \in 1..n}]$ consists of a method *new* and encodings for the methods as *pre-methods* – functions from a receiving

[1] A variation of $Ob_{1<:}$ provides contravariant methods as well. These are not needed for our purposes, and we do not explicitly address them: however, there seems no *a priori* reason why they cannot be accomodated.

(Type Class)
$$\frac{\Gamma \vdash A}{\mathit{Class}(A)} \quad \text{object type } A$$

(Sub Class)
$$\frac{\Gamma \vdash \mathit{Class}(A) \qquad \mathit{Class}(B) \qquad A <: B}{\mathit{Class}(A) <: \mathit{Class}(B)}$$

Fig. 3. $Ob_{1<:}$ classes and class sub-typing

object type A to the method body. The covariance decorations ensure $A <: B \Rightarrow \mathit{Class}(A) <: \mathit{Class}(B)$ (figure 3). Each member of $\mathit{Class}(A)$ has the form $[\mathit{new} = \varsigma(z : \mathit{Class}(A))[l_i = \varsigma(\mathit{self} : A)z.l_i(s)^{i \in 1..n}], l_i = \lambda(s : A)a_i^{i \in 1..n}]$, where *new* constructs an object where each member invokes the appropriate pre-method on the class. The *new* method models the mechanical process of constructing an object from a class rather than being a "constructor" in the usual sense of object-oriented languages.

3 Ionic objects

In this section we develop a minimal type model for ions.

First we need to formalise some intuitions about the construction of objects. In many cases an object is composed of sets of methods, each implementing a particular aspect or feature of the object's overall functionality. These features are often largely independent but weakly interacting. In some cases it is desirable to adapt the object by replacing one feature without disturbing the others, or extracting one or more features for use elsewhere. It is these manipulations that ionic types seek to model.

Let $A \equiv [l_i : A_i^{i \in 1..n}]$ be an object type having members labeled from $L_A \equiv \{l_i^{i \in 1..n}\}$. We may partition this label set into $\{L_J\}_{J \subseteq \{1..n\}}$, each of which induces an object type $O_J \equiv [l_j : A_j^{j \in J}]$. Each O_J defines a feature which combine to form the final definition of O. Generally we say that the O_J *cover* A (or equivalently that the L_J cover L_A), since $A <: O_J$ for all the O_J with no extra methods being introduced. We also introduce a type extension operator. If $A \equiv [l_i : A_i^{i \in 1..n}]$ and $B \equiv [l_j : B_j^{j \in n+1..m}]$ are object types, we define $A \triangleright B \equiv [l_i : A_i^{i \in 1..n}, l_j : B_j^{j \in n+1..m}]$ as extending A with the members of B. Note that in this definition the two types share no common labels, so both $A \triangleright B <: A$ and $A \triangleright B <: B$.

An object is composed of methods, which in general depend on the existence of other methods. An ion is created by removing some of these methods, giving a "proto-object" which may only be used again when these methods have been replaced. An application constructs an ion by specifying those methods of an object which should be retained in the ion; the type rules track the "missing" methods to guarantee that they are replaced by compatible implementations when the ion is de-ionised back to an object (figure 4).

Let two label sets L_I and L_D induce types I and D covering A. Suppose we have an object $a : A$, and we wish to capture the members in L_I. We cannot simply remove those members in L_D as the bodies of the L_I members will generally depend on the L_D for their definition. We therefore capture the dependency which the members in L_I have on the members in L_D by defining a

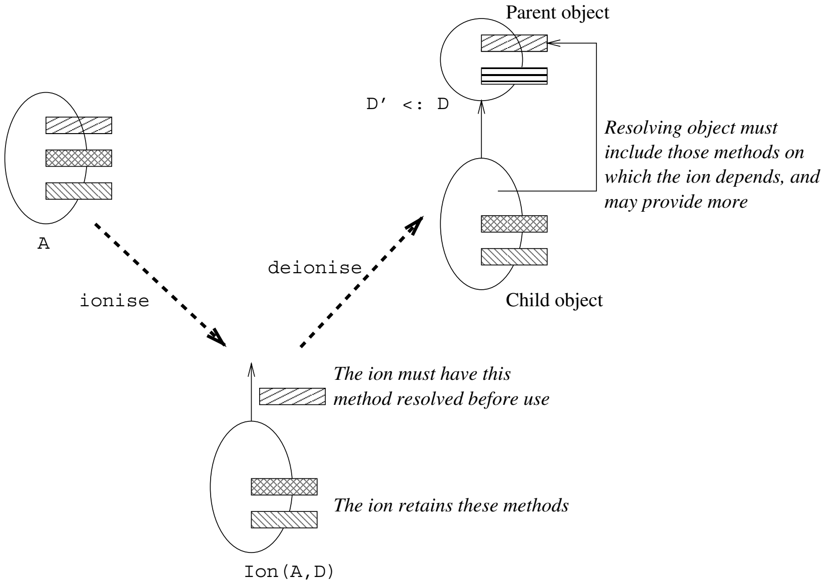

Parent object
D′ <: D
Resolving object must include those methods on which the ion depends, and may provide more
A
deionise
ionise
Child object
The ion must have this method resolved before use
The ion retains these methods
Ion(A,D)

Fig. 4. Ion basics

type $Ion(A, D)$ (read as "ion of A at D"). Because all labels are unique, A and D uniquely specify I, and also $A <: D$. The members of such types – the *ions* – represent objects which have some of their members abstracted and which may be replaced by other suitably-typed members. If L_D has no members – so $D \equiv []$, the empty object type usually referred to as *root* – then the ion has no dependencies and is essentially an ordinary object[2]. We refer to A as the *underlying type* of the ion, I as its *ionisation* type, and D as its *dependency type.*

The type rules for ions are shown in figure 5. Ionic types are covariant in their members and contravariant in their dependencies: for an ion i' to be substitutable for an ion i it must provide *at least* the same members with *no more* dependencies. The $ionise()$ operation generates an ion from an object given a description of the methods to retain. The $deionise()$ operation resolves these dependencies against another suitably-typed object, delegating the methods in the dependency type to this object and yielding an object which includes the actual type of the object against which the ion is resolved.

There is a small subtlety concerning method overriding. Suppose that we have $i : Ion(A, D)$ and $d : D' <: D$ where D' also includes a member $l_j : A_j^{j \in L_I}$. It is possible that both A and D' may include a member l_j, so $deionise(i, d)$ must select which method body to use in the deionised object. It is also possible that the two method types are incompatible, since A and D' may be sub-types

[2] One could therefore define an object calculus completely in terms of ions, or equate $Ion(A, [])$ with A.

(Type Ion)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash I \quad \Gamma \vdash D}{\Gamma \vdash Ion(A, D)} \text{ object type } A \text{ covered by } I, D$$

(Sub Ion)

$$\frac{\Gamma \vdash Ion(A, D) \quad \Gamma \vdash Ion(A', D') \quad \Gamma \vdash A' <: A \quad \Gamma \vdash D <: D'}{\Gamma \vdash Ion(A', D') <: Ion(A, D)}$$

(Val Ionise)

$$\frac{\begin{array}{c}\Gamma \vdash A \quad \Gamma \vdash I \quad \Gamma \vdash D \\ \Gamma \vdash Ion(A, D) \quad \Gamma \vdash a : A\end{array}}{\Gamma \vdash ionise(a, I) : Ion(A, D)} \text{ object type } A \text{ covered by } I, D$$

(Val Deionise)

$$\frac{\begin{array}{c}\Gamma \vdash A \quad \Gamma \vdash I \quad \Gamma \vdash D \\ \Gamma \vdash i : Ion(A, D) \quad \Gamma \vdash D' <: D \quad \Gamma \vdash d : D'\end{array}}{\Gamma \vdash deionise(i, d) : I \rhd D'} \text{ object type } A \text{ covered by } I, D$$

Fig. 5. Type rules for ions

of D down two different sub-typing paths. Since both i and d may contain a definition of l_j we must make three decisions:

1. If a method defined in i invokes l_j, which implementation is used?;
2. If a method defined in d invokes l_j, which implementation is used?; and
3. If a client of $deionise(i, d)$ invokes l_j, what implementation is used?

This problem has arisen in other guises, and most of the combinations have been tried (see section 7). In the current application, we want to use ions to model transparent delegation of functionality to other objects in order to support mobility and other applications. Specifically we want to be able to deionise several ions against the same parent object whilst leaving the parent's behaviour unchanged. This contrasts with most delegation-based languages in which the additional functionality may change the behaviour of the parent object.

We therefore adopt the following model:

1. If the method is defined only in i, invocations from clients of $deionise(i, d)$ and from methods defined in i use the implementation from i – by definition there will be no invocations from d;
2. If the method is defined in i and d, and the two are type-incompatible, a dynamic type error occurs;
3. Otherwise, all invocations use the implementation from d.

It should be noted that point 2 is the only place where ionic types require dynamic checks.

These rules are formalised in figure 6. (The side conditions reflect the interactions between ions and classes dealt with in the next section.) The object d is left completely unaffected by deionisation against it, meaning that several child objects may safely be deionised against a single parent without sensitivity to order. However, some of the ion's functionality may not be propagated into the final deionised object. This is the critical difference between ions and delegation-based systems: they prevent overriding using the type system. It means that an ion can only add functionality to another object, not affect functionality already present. This is essential for modeling mobility using ions: if code from the network is allowed to override system-provided functionality, it opens a gaping security hole. These issues are examined further in section 5.

(Eq Ionise)

$$\frac{\Gamma \vdash a \leftrightarrow [l_j = a_j^{l_j \in L_I \cup L_D}] : A}{\Gamma \vdash ionise(a, I) \leftrightarrow [l_j = a_j^{j \in L_I}] : Ion(A, D)} \quad L_I, L_D \text{ cover } L_A, A, I \text{ not class types}$$

(Eq Deionise)

$$\frac{\begin{array}{c}\Gamma \vdash c \leftrightarrow [l_j = c_j^{l_j \in L_I}] : Ion(A, D) \\ \Gamma \vdash d \leftrightarrow [l_k = d_k^{l_k \in L_D \cup L_X}] : D'\end{array}}{\Gamma \vdash deionise(c, d) \leftrightarrow [l_j = c_j^{l_j \in L_I - L_X}, l_k = d_k^{l_k \in L_D \cup L_X}] : I \rhd D'} \quad A, D \text{ not class types}$$

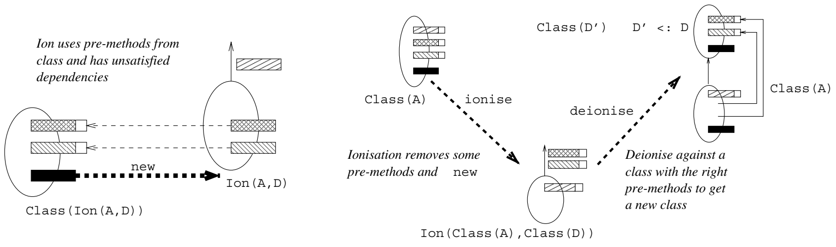
Fig. 6. Semantics for ions

A further consequence of the rule (Val Deionise) in figure 5 is that the final type of a child object resulting from deionisation reflects the actual type of its parent: the final object is a valid sub-type of the ionisation type, the actual type of the parent object, and (by subsumption) the dependency type. This means that an application receiving an ion may both add the ion's functionality to an object and ensure that the object's extra methods (over and above those in the ion's dependency type) remain accessible. Assuming no name clashes, these extra methods are invisible to the method bodies derived from the ion.

4 Ions and classes

As mentioned in section two, $Ob_{1<:}$ does not treat classes as primitive. Rather, a class is an object which produces objects of a given underlying object type which share a common implementation of their methods. If X is an object type then $Class(X)$ is the object type of classes generating objects of type X. In this section we explore the interactions of the class and ion constructions.

The first observation is that the class construction may be applied to ionic types in two distinct ways. The first builds a family of types $Class(Ion(A, D))$, representing classes which create ions rather than objects. The second builds a family of types $Ion(Class(A), Class(D))$, representing classes with some functionality extracted. We refer to the former as *ion classes* and the latter as *ionised classes*[3].


Ion uses pre-methods from class and has unsatisfied dependencies
new
Ion(A,D)
Class(Ion(A,D))
Class(A)
ionise
Ionisation removes some pre-methods and new
Ion(Class(A),Class(D))
Class(D')
D' <: D
deionise
Deionise against a class with the right pre-methods to get a new class
Class(A)

(a) Ion class

(b) Ionised class

Fig. 7. Ion and ionised classes

An ion class constructs ions in the same way as a class constructs objects: it generates instances of an object type which share a common implementation (figure 7a). Each object created by a class is identical to all others in terms of initial state but independent in terms of future state transitions; similarly the ions created by an ion class are initially identical but may be deionised independently to generate objects which will be functionally distinct despite their common heritage.

An ion class may be encoded in a manner very similar to the usual class encoding. The pre-methods specify the defined parts of the ion as expected, but the constructor returns an ion including some dependencies. Suppose $A \equiv [l_i : A_i^{i \in 1..n}]$, L_D and L_I cover L_A, and $D \equiv [l_i : A_i^{l_i \in L_D}]$. Then $Class(Ion(A, D)) \equiv [new^+ : () \to Ion(A, D), l_i^+ : A \to A_i^{i \in L_I}]$ – the pre-methods map an object of the underlying type to the provided body.

Ionised classes are a little more subtle, requiring explicit handling of *new* methods. Ionising a class removes some of its pre-methods and *new*: the resulting object is then deionised against a class providing the abstracted pre-methods (figure 7b). Deionisation synthesises a *new* method reflecting the new class' structure (figure 8). Although the rule (Eq Class Deionise) looks complex, it basically just performs sequential composition of the *new* methods with duplicates being overridden by methods from d as expected.

It is interesting to note the difference between an ionised class and an abstract class. In an abstract class some of the class' methods are left to be defined by the sub-classes; in an ionised class some of the methods are left to be defined by the dependency type, which is effectively an abstracted superclass. This inversion of the normal extension approach means that a designer may apply *the same* extended functionality to *different* leaves in the inheritance hierarchy, whereas

[3] There are actually two other possibilities, $Ion(Class(A), D)$ and $Ion(A, Class(D))$, which do not have obvious interpretations for arbitrary types A and D.

(Eq Class Ionise)

$$\frac{\mathit{\Gamma} \vdash a \leftrightarrow [\mathit{new} = \ldots, l_i = \lambda(s : A)a_i^{l_i \in L_A}] : \mathit{Class}(A)}{\mathit{\Gamma} \vdash \mathit{ionise}(a, I) \leftrightarrow [l_i = \lambda(s : A)a_i^{l_i \in L_I}] : \mathit{Ion}(\mathit{Class}(A), \mathit{Class}(D))} \; L_I, L_D \text{ cover } L_A$$

(Eq Class Deionise)

$$\frac{\begin{array}{c} \mathit{\Gamma} \vdash c \leftrightarrow [l_i = c_i^{l_i \in L_I}] : \mathit{Ion}(\mathit{Class}(A), \mathit{Class}(D)) \\ \mathit{\Gamma} \vdash d \leftrightarrow [\mathit{new} = \ldots, l_j = d_j^{l_j \in L_{D'}}] : \mathit{Class}(D') \qquad \mathit{\Gamma} \vdash D' <: D \end{array}}{\begin{array}{c} \mathit{\Gamma} \vdash \mathit{deionise}(c, d) \leftrightarrow \\ \left[\begin{array}{c} \mathit{new} = \varsigma(z : \mathit{Class}(I \rhd D'))[l_i = \varsigma(\mathit{self} : I \rhd D')z.l_i(\mathit{self})^{l_i \in L_I \cup L_{D'}}], \\ l_i = \lambda(s : I \rhd D')c_i^{l_i \in L_I - L_{D'}}, \\ l_i = \lambda(s : I \rhd D')d_i^{l_i \in L_{D'}} \end{array}\right] : \text{Class(I } \rhd D') \end{array}}$$

Fig. 8. Ionisation rules for class types

the normal inheritance approach would require such shared functionality to be applied to a common ancestor of the leaves – which is typically not possible in systems which are dynamically extended. Ionised classes therefore provide an important foil to the normal inheritance mechanism.

The traditional approach to controlling overriding has been through visibility modifiers such as Java's `public`, `protected` and `final` modifiers. Such modifiers define a single overriding policy for all the code in a system – and when code increasingly derives from multiple sources with different degrees of trust this seems to be a very blunt implement with which to tackle an increasingly subtle problem. With ionised classes, rather than extending a single class under a single policy, we may apply the same extension to different bases. This adds a layer of flexibility when (for example) the extensions are imported from remote sources, since the base may be tailored to suit the source.

5 Applications

In this section we show how ions and ionised classes can be used to model and improve four common programming structures.

5.1 Method update

Object calculi (including $Ob_{1<:}$) frequently allow the methods of an object to be updated as computation progresses. In a formal system this allows value and method update to be handled uniformly; in practice it can also be a useful capability, for example in providing dynamic adaptation of key algorithms. Since mainstream languages do not support method update, such features are usually addressed using (unchecked) design patterns. Ions provide a direct solution, with the additional benefit of being able to perform multiple updates simultaneously.

The static, inheritance-based approach to method update defines a sub-class with the new method implementations, adding the new implementations from the bottom of the inheritance hierarchy. The dynamic case using ions does the opposite: the object is first ionised to remove the unwanted implementations, and is then deionised against an object providing the new implementations. Since deionisation must satisfy all abstracted dependencies, one may force a set of simultaneous updates simply by controlling the initial ionisation.

This definition of method update retains the old object, dynamically generates a copy with the new functionality, and does not disturb any other objects – a *run-time, selective, preserving* adaptation to use the terminology of [13]. The same approach may be applied to classes, building an ionised class without the unwanted functionality and deionising against a class providing the new implementations. The preservation of the old definition may be undesirable in some circumstances, as it may be impossible to transparently replace the old definition with the new throughout a complex program.

5.2 Agents

An agent is an object instance which moves between sites according to some itinery. The agent may retain state gathered at each site and possibly use it to determine its future path. At each site the agent requires access to some part of the host site's environment, and the capabilities which each site grants will typically depend on the agent's source. Agents are usually represented as objects, but their environmental dependence means that they are more correctly viewed as ions.

When an agent migrates, the host will not typically transfer the entire program: instead it will transfer that part of the program which provides the agent's novel functionality, assuming that the common parts of the run-time system will be made available by the receiving host. This assumption is flawed, in the sense that a host may decide not to make some sensitive functions available to some incoming agents. This in turn means that, if the agent depends on those functions, it cannot successfully execute. However, this dependence remains implicit within the agent rather than being made explicit in its type, making it difficult to determine *a priori* whether the agent has an unsatisfied dependency or not.

As an example, suppose we have an environment containing an operation which prints an integer on the screen, $Env \equiv [printint : int \rightarrow ()]$. This operation depends critically on location in terms of resource acquisition (different places use different displays); it is also security-sensitive, in that untrusted code may not be allowed to access the screen. This means that a host may wish not to make *printint* available to all agents.

Suppose we now define an object type $O \equiv Env \triangleright [i : int, inc : int \rightarrow (), print : () \rightarrow ()]$ representing an agent running in an environment and maintaining a counter which can be incremented and printed on the screen. The *print* method of O uses the *printint* method of Env to display the counter on the screen before moving to the next host.

To migrate o from one site to another we ionise it at Env to capture its environmental dependency, transport it using the appropriate wire protocol, and then deionise it against the appropriate local environment (figure 9).

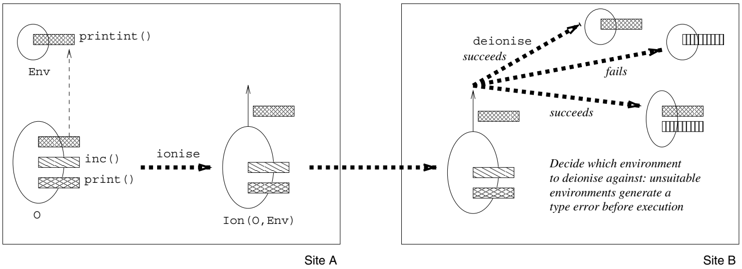


Fig. 9. Ions and migrating agents

If the destination site decides that the incoming *o* should not be allowed to access the screen, it may supply an environment without the *printint* method. Deionising *o* against this environment will fail with a type error without execution.

An application receiving an agent may, in addition to running it, wish to provide monitoring "hooks" to track its activities. Ionic types offer a simple way of providing these hooks. The application adds the monitoring methods to the environment (for example by defining an environment $Env' \equiv [printint : int \rightarrow (), numberofintsprinted : () \rightarrow int]$ with a hook to monitor output behaviour). Deionising against this extended environment allows the extra method to show through (a consequence of (Val Deionise) in figure 5), allowing the receiving application to access both the incoming code and its own monitoring and control code.

We may contrast the behaviour of ions against the fully dynamic approach where environmental access is provided by system-wide objects such as Java's use of `java.lang.System`. An incoming object's dependence on these objects is poorly captured, so operations which are known to be generally disallowed (such as file access) are only caught at run-time. Furthermore some of the objects may be unavailable even though they remain notionally defined (for example Java's `System.out` console stream, which can be a "black hole"), leading to unexpected behaviour.

By capturing explicitly the dependence which an object has on routines which potentially change across environments we provide a hook for improved analysis and control. One may, for example, define the minimal dependence which an agent has on its environment as the dependency type of the ion, and then analyse precisely the impact of different security regimes.

5.3 Applets

Java applets are a common form of mobile code. The main difference between an applet and an agent is that, while agent systems move objects, applets move classes which are instantiated afresh at each site.

Applets generally run in an environment with very restricted disk and network access. These restrictions are enforced by run-time checks inserted manually into all “sensitive” methods, allowing a browser to intercept and disallow calls which might violate security. For many systems the restrictions will simply disallow all accesses to some methods by applets, throwing a security exception if the method is called. This simple form of security is basically a type-checking problem, since a call to such a method will *always* fail.

We may model this situation using an ionised class. Rather than treat the applet as a fully-fledged class – implying that its execution environment is the same as that in which it was defined – we instead ionise the applet class before transporting it to the client-side. The ionised applet class is then deionised against an environment including exactly those methods which the applet is allowed to call. If the applet requires more methods then this environment will not completely satisfy its dependency type and the deionisation operation will fail before the applet is executed. This prevents partial completions whose consequences may range from annoying to catestrophic.

This example illustrates the use of ionic types to capture, within the type system, what are effectively load-time link errors. It provides a useful extra “sanity check” for all dynamically loaded classes, not just applets.

5.4 Delegation

Delegation is often seen as an alternative to inheritance when building object-oriented programs. In inheritance, the methods of an object are composed by adding new members to an already-existing class, which is then instantiated. In delegation, some of the methods of one object are passed-off to be handled by another object. Delegation therefore involves two objects rather than one. In most mainstream object-oriented languages delegation is treated as a design pattern rather than as a basic language construct, and must be constructed *ad hoc* by each application.

The simplest case is where a child object delegates to a parent without affecting its behaviour – adding methods but not overriding any. We define an ion i whose underlying type include all the available methods and whose dependency type contains the signatures of the methods to be delegated. We then deionise against the object d to which those methods should be delegated. The fact that d is self-contained in $deionise(i, d)$ – so the methods of d remain unaffected by the addition of i – means that it is perfectly safe to delegate several ions (potentially with different underlying types or extended implementations) to the same object.

Ions are not a complete replacement for general delegation, however. A general delegation system allows the child object to override methods in the parent. If there are multiple children, the parent must select the method definition from the child on which the invocation originated – the so-called “self problem”[14]. Ions explicitly outlaw this form of behavioural modification – the child’s methods cannot override the parent’s – so ion-based delegation is only applicable when *extending* the parent’s functionality.

Delegation is particularly important in distributed systems, where an object may wish to extend the functionality of another object on a remote server.

Distributed object systems such as CORBA or RMI generate "stub" objects to act as local representatives of remote services. Both inheritance and delegation are problematic in these situations, as the client may attempt to override some of the server's functionality without the server being notified. This means that, while calls to the client will use the overridden method, embedded calls within the server will use the original implementation.

Ions may be used to provide "wrapper" objects around the stub without introducing these problems. The client constructs an ion with the added functionality and deionises it against the stub. The semantics of ions means that the server's behaviour remains unchanged, so the resulting object behaves exactly as expected if the object were local instead of being represented by a stub. This makes it trivially easy for a local object to delegate some of its functionality to a remote object with no "surprises".

6 Ionic types in Java

How might one add ionic types to Java? There are several possible solutions, and in this section we sketch perhaps the simplest approach which extends Java minimally (in terms of syntax, semantics and "feel") to provide ionic constructs. The presentation is informal, our intention being to indicate the feasibility of providing ions in Java rather than to fully formalise and extend Java's type system – a task already well performed by others[9].

Java's designers have consistently maintained the significance of the names given to classes and interfaces, as well as their structure. This suggests a model of ions where we use interfaces to indicate allowed ionisation boundaries. This respects Java's insistence that all class and objects types are named, which is a little more restrictive than the arbitrary ionisation allowed by our presentation so far. It turns out that this decision also has a slight impact on the definition of deionisation.

Consider a Java declaration `class A implements D` { `...` } together with an instance `a`. We may ionise `a` at the `D` interface boundary to construct an ion `i` having an extended Java type `ion A at D` corresponding informally to our $Ob_{1<:}$ type $Ion(A, D)$ and represented as a clone of `a` with the methods from `D` abstracted. We may then deionise `i` against an object `b` of class `B` implementing interface `D` in the expected way. Operationally this delegates the `D`-declared methods of `i` to `b`. The definition of `class A` may include other interfaces as well as `D`, which should be preserved by ionisation.

What is the type of the deionised object? According to our rules it extends `A` with the methods of `B` – but this type is novel, synthetic and without a name, meaning that it cannot be expressed in a Java program. However, we *can* guarantee that it is a legal instance of `A`, `B` or `D`, which is sufficient to allow it to be used.

An ion class is declared by providing the definition of its methods assuming that it implements the methods of the interface along which it is ionised. Creating an instance of this class constructs an ion, which may then be deionised to

```
class A implements D { ... }
class B implements D { ... }
A a = new A();   B b = new B();
ion A at D i = ionise a at D;
A aprime = deionise i against b;
```

Fig. 10. Ionising Java objects

form an object (figure 11, top). Ionised classes are a bit more subtle, as they effectively allow run-time manipulation of the class hierarchy. We introduce a new Java type `ion class A at D` representing a class `A` with methods declared in `D` abstracted, creating such objects using `ionise class A at D` and allowing them to be passed around and resolved (figure 11, bottom). Visibility modifiers behave as expected: a `private` modifier on the base prevents an ion depending on that method, and so forth. Composition of constructors is similarly well-behaved.

```
class ion A at D { ... }
ion A at D a = new A();

ion class A at D Aprime = ionise class A at D;
A a = new (deionise Aprime against class B)();
```

Fig. 11. Ion and ionised classes in Java

How do these constructs support Java-based mobile code systems? We need the ionic operations to preserve special interfaces such as `java.io.Serializable`, so that the ion is serialisable if the object is. Similarly we want ionised classes to be serialisable so that they may be passed between machines. This is possible in Java – outside the type system! – using `Class` objects and the reflection API, so the ionic operations may be regarded simply as giving a typed basis to existing low-level functionality.

This sketch of ions in Java remains incomplete, leaving a number of thorny issues inevitable when trying to integrate a new construct into an existing language – most notably the impact of threads, class loaders, and the handling of interfaces such as `java.io.Externalizable` which could generate subtle errors if applied to ions. We hope we have indicated the ions lie sufficiently close to modern practice to be a viable proposition.

7 Related work

The ability to add functionality uniformly to the bottom of a class hierarchy – usually referred to as *mixin-based* inheritance – has been studied extensively, *e.g.* [4][10]. Mixins arose within a dynamic typing regime, and have been extended with static checks. Ionised classes offer essentially the same functionality as mixins, and are statically type-safe in environments where all classes are elaborated at compile-time. The more general ionic types offer more dynamism with a small and controlled loss of type safety.

The problem of name capture and overriding discussed in section 3 can be resolved in several different ways. Delegation-based languages generally allow the child object to override methods in the parent, changing its behaviour. (An elegant solution to name capture along different sub-typing paths in this context is given by Kniesel[13].) The Beta language[15] prohibits simple overriding but allows a child method to be called within the parent method. Neither approach effectively addresses untrusted downloaded code or transparent delegation to remote objects.

Many groups are working towards the design and implementation of mobile agent systems – a good review may be found in [12]. Most of this work has focussed on mechanisms for migration (*e.g.* [11]) and on traditional authorisation and verification approaches such as digital signatures, assuming that an agent's environment is in some senses the same across hosts. Treating environmental changes as a phenomenon in their own right provides a different – and in some ways more flexible – view of these issues, allowing them to be treated more uniformly within a type framework. Another substantial body of work has addressed low-level aspects of mobile systems such as channel usage[17] and encrypted transport protocols[3] using type systems. These efforts may be seen as part of a trend which replaces *ad hoc* checks with richer type checking.

The ambient calculus of Cardelli and Gordon[6] is a particularly interesting emerging formalism for mobility. An ambient encapsulates a set of processes and sub-ambients into a package which may migrate through a hierarchy according to some simple rules. Two type frameworks have been developed on top of the original untyped system: a conventional system controlling the types of communications[7] and a novel system of "mobility types"[5] providing type-level control over how an ambient may move about the system. We speculate that communication types are closely related to ions, in the sense that an ion represented as an ambient might add new communications (method calls) if opened within another context. This is a possible area for future work.

Work on design patterns offers a great many useful approaches to designing complex systems. It is interesting to note that many patterns simply provide mechanisms for controlling behavioural updates and indirections – issues which may in many cases be addressed type-safely using ions. This addresses a major criticism of design patterns, that they add layers of complexity to applications without providing additional layers of checking.

8 Conclusion

We have described a minimal model of ionic types and shown how they can model some otherwise problematic issues in systems with dynamic code completion. We showed some applications of the technique to mobile code, simple security checking, delegation and method update. The uniform treatment provides suitable type guarantees to many aspects usually addressed using design patterns or fully dynamic type-checking. We briefly sketched an incomplete embedding of ions into Java, suggesting that the concepts could easily be reified in a practical programming language.

Existing models of object-oriented systems are designed with a static, monolithic system in mind. With distributed systems, mobile code and policy-controlled access becoming more commonplace, these models are no longer sufficiently powerful to represent all the desirable behaviours. The main contribution of this work is to provide a simple model for structured behavioural change which accurately reflects the capabilities, limitations and threats observed in the emerging generation of systems and which allows static and dynamic modifications to be treated uniformly.

Type checking evolved as a way of avoiding as many run-time "sanity checks" as possible. As application domains become more complex it becomes vital to encompass additional properties within the type system, avoiding as far as possible an explosion in avoidable run-time checks (and, by implication, avoidable run-time failures). New versions of Java, for example, will allow applications to define security domains for incoming code, meaning that the behaviour of a migrating object may be radically affected as it changes domains[1]. One may use ions to model the static exclusions necessary for a policy, highlighting those methods which need context-sensitive handling. This can suggest different design approaches, for example separating potentially sensitive functions which manipulate files from their more general counterparts. We are also exploring refinements by which the dependency type of an ionic type can be expanded and reduced piecemeal, to provide finer-grained control over ion dependencies.

A number of implementations of ions are possible, and we have sketched a restricted model close to the spirit of Java. We are exploring other models using the Vanilla language toolkit[8] with a view to determining the extent to which ions provide a widely applicable orthogonal type construction to be integrated into languages for mobile computing. We are also interested in the interaction of ions with native compilation, especially systems such as Harissa[16] which are able to handle dynamic class loading.

References

1. Secure computing with Java: now and the future. http://www.javasoft.com/marketing/collateral/security.html, 1998.
2. Martín Abadi and Luca Cardelli. *A theory of objects*. Springer Verlag, 1996.
3. Martín Abadi and Andrew Gordon. Secrecy by typing in security protocols. In *Theoretical aspects of computer science*, volume 1243, pages 59–73. Springer-Verlag, 1997.
4. Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of OOPSLA/ECOO'90*, 1990.
5. Luca Cardelli, Giorgio Ghelli, and Andrew Gordon. Mobility types for mobile ambients. In *Proceedings of the International Conference on Algebraic and Logic Programming*, 1999.

6. Luca Cardelli and Andrew Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of software science and computational structures*, volume 1378. Springer Verlag, 1998.
7. Luca Cardelli and Andrew Gordon. Types for mobile ambients. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 79–92, 1999.
8. Simon Dobson, Paddy Nixon, Vincent Wade, Sotirios Terzis, and John Fuller. Vanilla: an open language framework. In Krzysztof Czarnecki and Ulrich Eisenecker, editors, *Generative and component-based software engineering*. Springer-Verlag, 1999.
9. Sophia Drossopoulou, Susan Eisenbach, and Sarfraz Khurshid. Is the Java type system sound? *Theory and practice of object systems*, 5(1):3–24, 1999.
10. Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of POPL'98*, pages 171–183. ACM Press, 1998.
11. Eric Jul, Henry Levy, Neil Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.
12. Neeran Karnik and Anand Tripathi. Design issues in mobile-agent programming. *IEEE Concurrency*, 6(3), 1998.
13. Günther Kniesel. Type-safe delegation for run-time component adaptation. In *Proceedings of ECOOP'99*, pages 351–366. Springer-Verlag, 1999.
14. Henry Lieberman. Using prototypical objects to implement shared behaviour in object-oriented systems. In *Proceedings of OOPSLA'86*, 1986.
15. Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-oriented programming in the BETA programming language.* Addison-Wesley, 1993.
16. Gilles Muller and Ulrik Pagh Schultz. Harissa: a hybrid approach to Java execution. *IEEE Software*, 16(2):44–51, 1999.
17. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993.

Load-Time Structural Reflection in Java

Shigeru Chiba

Institute of Information Science and Electronics
University of Tsukuba
and Japan Science and Technology Corp.
chiba@is.tsukuba.ac.jp, chiba@acm.org

Abstract. The standard reflection API of Java provides the ability to introspect a program but not to alter program behavior. This paper presents an extension to the reflection API for addressing this limitation. Unlike other extensions enabling behavioral reflection, our extension called *Javassist* enables structural reflection in Java. For using a standard Java virtual machine (JVM) and avoiding a performance problem, Javassist allows structural reflection only before a class is loaded into the JVM. However, Javassist still covers various applications including a language extension emulating behavioral reflection. This paper also presents the design principles of Javassist, which distinguish Javassist from related work.

1 Introduction

Java is a programming language supporting reflection. The reflective ability of Java is called the reflection API. However, it is almost restricted to introspection, which is the ability to introspect data structures used in a program such as a class. The Java's ability to alter program behavior is very limited; it only allows a program to instantiate a class, to get/set a field value, and to invoke a method through the API.

To address the limitations of the Java reflection API, several extensions have been proposed. Most of these extensions enable behavioral reflection, which is the ability to intercept an operation such as method invocation and alter the behavior of that operation. If an operation is intercepted, the runtime systems of those extensions call a method on a *metaobject* for notifying it of that event. The programmer can define their own version of metaobject so that the metaobject executes the intercepted operation with customized semantics, which implement a language extension for a specific application domain such as fault tolerance [9].

However, behavioral reflection only provides the ability to alter the behavior of operations in a program but not provides the ability to alter data structures used in the program, which are statically fixed at compile time (or, in languages like Lisp, when they are first defined). The latter ability called structural reflection allows a program to change, for example, the definition of a class, a function, and a record on demand. Some kinds of language extensions require this ability for implementation and thus they cannot be implemented with a straightforward

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 313–336, 2000.
© Springer-Verlag Berlin Heidelberg 2000

program using behavioral reflection; complex programming tricks are often needed.

To simply implement these language extensions, this paper presents *Javassist*, which is a class library for enabling structural reflection in Java. Since portability is important in Java, we designed a new architecture for structural reflection, which can be implemented without modifying an existing runtime system or compiler. Javassist is a Java implementation of that architecture. An essential idea of this architecture is that structural reflection is performed by bytecode transformation at compile-time or load time. Javassist does not allow structural reflection after a compiled program is loaded into the JVM. Another feature of our architecture is that it provides source-level abstraction: the users of Javassist do not have to have a deep understanding of the Java bytecode. Our architecture can also execute structural reflection faster than the compile-time metaobject protocol used by OpenC++ [3] and OpenJava [20].

In the rest of this paper, we first overview previous extensions enabling behavioral reflection in Java and point out limitations of those extensions. Then we present the design of Javassist in Section 3 and show typical applications of Javassist in Section 4. In Section 5, we compare our architecture with related work. Section 6 is conclusion.

2 Extensions to the Reflection Ability of Java

The Java reflection API dose not provide the full reflective capability. It does not enable alteration of program behavior but it only supports introspection, which is the ability to introspect data structures, for example, inspecting a class definition. This design decision was acceptable because implementing the full capability was difficult without a decline in runtime performance. An implementation technique using partial evaluation has been proposed [17,2] but the feasibility of this technique in Java has not been clear.

However, several extensions to the Java reflection API have been proposed. To avoid performance degradation, most of these extensions enable restricted behavioral reflection. They only allow alteration of the behavior of specific kinds of operations such as method calls, field accesses, and object creation. The programmers can select some of those operations and alter their behavior. The compilers or the runtime systems of those extensions insert *hooks* in programs so that the execution of the selected operations is intercepted. If these operations are intercepted, the runtime system calls a method on an object (called *a metaobject*) associated with the operations or the target objects. The execution of the intercepted operation is implemented by that method. The programmers can define their own version of metaobject for implementing new behavior of the intercepted operations.

The runtime overheads due to this restricted behavioral reflection are low since only the execution of the intercepted operations involves a performance penalty and the rest of the program runs without any overheads. Especially, if hooks for the interception are statically inserted in a program during compilation, the runtime overheads are even lowered. To statically insert hooks, Reflective Java [22] performs source-to-source translation before compilation and Kava [21] performs bytecode-level transformation when a program is loaded into the JVM. MetaXa [16,11] internally performs bytecode-level transformation with a customized JVM. It uses a customized just-in-time compiler (JIT) for improving the execution speed of the inserted hooks. This hook-insertion technique is well known and has been applied to other languages such as C++ [4].

Although the restricted behavioral reflection is useful for implementing various language extensions, there are some kinds of extensions that cannot be intuitively implemented with that kind of reflection. An example of these extensions is binary code adaptation (BCA) [13], which is a mechanism for altering a class definition in binary form to conform changes of the definitions of other classes. Suppose that we write a program using a class library obtained from a third party. For example, our class Calendar implements an interface Writable included in that class library:

```
class Calendar implements Writable {
  public void write(PrintStream s) { ... }
}
```

The class Calendar implements method `write()` declared in the interface Writable.

Then, suppose that the third party gives us a new version of their class library, in which the interface Writable is renamed into Printable and it declares a new method `print()`. To make our program conform this new class library, we must edit the definitions of all our classes implementing Writable, including Calendar:

```
class Calendar implements Printable {
  public void write(PrintStream s) { ... }
  public void print() { write(System.out); }
}
```

The interface of Calendar is changed into Printable and method `print()` is added.

BCA automates this adaptation; it automatically alters class definitions in binary form according to a configuration file specifying how to alter them. Note that the method body of `print()` is identical among all the updated classes since `print()` can be implemented with the functionality already provided by `write()` for the old version. If that configuration file is supplied by the library developer, we can run our program without concern about evolution of the class library.

Unfortunately, implementing BCA with behavioral reflection is not intuitive or straightforward. Since behavioral reflection cannot directly provide the ability to alter data structures such as a class definition or construct a new data structure, these reflective computation must be indirectly implemented. For example, the implementation of BCA with behavioral reflection defines a metaobject indirectly performing the adaptation specified by a given configuration file. For the above example, this metaobject is made to be associated with Calendar and it watches method calls on Calendar objects. If the method print() is called, the metaobject intercepts that method call and executes the computation corresponding to print() instead of the Calendar object. The metaobject also intercepts runtime type checking so that the JVM recognizes Calendar as a subtype of Printable. Recall that Java is a statically typed language and the original Calendar is a subtype of Writable.

The ability to alter data structures used in a program is called structural reflection, which has not been directly supported by previous systems. Although a number of language extensions are more easily implemented with structural reflection than with behavioral reflection, the previous systems have not been addressing those extensions. They have been too much focused on language extensions that can be implemented by altering the behavior of method calls and so on.

3 Javassist

To simply implement language extensions like BCA shown in the previous section, we developed Javassist, which is our extension to the Java reflection API and enables structural reflection instead of behavioral one. Javassist is based on our new architecture for structural reflection, which can be implemented without modifying an existing runtime system or a compiler.

3.1 Implementations of Structural Reflection

Structural reflection is the ability to allow a program to alter the definitions of data structures such as classes and methods. It has been provided by several languages such as Smalltalk [10], ObjVlisp [6], and CLOS [14]. These languages implement structural reflection with support mechanisms embedded in runtime systems. Since the runtime systems contain internal data representing the definitions of data structures such as a class, the support mechanisms allow a program to directly read and change those internal data and thereby execute structural reflection on the correspondent data structures.

We could not accept this implementation technique for Javassist since it needs to modify a standard JVM but portability is important in Java. Furthermore, a naive application of this technique to Java would cause serious performance degradation of the JVM because this technique makes it difficult for runtime systems to employ optimization techniques based on static information of executed programs. Since a program may be altered at runtime, efficient dynamic recompilation is required for redoing optimization on demand. For example, method inlining is difficult to perform. If an inlined method is altered at runtime with structural reflection, all the inlined code must be updated. To do this, the runtime system must record where the code is inlined. This will spend a large amount of memory space. Another example is the "v-table" technique used for typical C++ implementations [8]. This technique statically constructs method dispatch tables so that invoked methods are quickly selected with a constant offset in the tables. If a new method is added to a class at runtime, then the dispatch tables may be updated and all offsets in the tables may be recomputed. Since the dynamic recompilation technique has been used so far for gradually optimizing "hot spots" of compiled code at runtime [12], it has been assuming that a program is never changed at runtime. Effectiveness of dynamic recompilation without this assumption is an open question.

Another problem is correctness of types. Since Java is a statically typed language, a variable of type X must be bound to an object of X or a subclass Y of X. If a program can freely access and change the internal data of the JVM, it may dynamically change the super class of Y from X to another class. This change causes a type error for the binding between a variable of type X and an object of Y. To address this problem, extra runtime type checks or restrictions on the range of structural reflection are needed.

3.2 Load-Time Structural Reflection

To avoid the problems mentioned above, we designed a new architecture for structural reflection; it does not need to modify an existing runtime system or a compiler. On the other hand, it enables structural reflection only before a program is loaded into a runtime system, that is, at load time. Javassist is a class library enabling structural reflection based on this architecture. In Java, the bytecode obtained by compilation of a program is stored in *class files*, each of which corresponds to a distinct class. Javassist performs structural reflection by translating alterations by structural reflection into equivalent bytecode transformation of the class files. After the transformation, the modified class files are loaded into the JVM and then no alterations are allowed after that. Thereby, Javassist can be used with a standard JVM, which may use various optimization techniques.

Javassist is used with a user class loader. Java allows programs to define their own versions of class loader, which fetch a class file from a not-standard resource such as a network. A typical definition of the class loader is as follows:

```
class MyLoader extends ClassLoader {
  public Class loadClass(String name) {
    byte[] bytecode = readClassFile(name);
    return resolveClass(defineClass(bytecode));
  }

  private byte[] readClassFile(String name) {
    // read a class file from a resource.
  }
}
```

The methods `defineClass()` and `resolveClass()` are inherited from ClassLoader. They request the JVM to load a class constructed from the bytecode given as an array of `byte`. The returned value is a Class object representing the loaded class. Once a class X is manually loaded with an instance of MyLoader, all classes referenced by that class X are loaded through that class loader. The

JVM automatically calls `loadClass()` on that class loader for loading them on demand.

Javassist helps `readClassFile()` shown above obtain the bytecode of a requested class. It can be regarded as a class library for reading bytecode from a class file and altering it. However, unlike similar class libraries such as the JavaClass API [7] and JOIE [5], Javassist provides source-level abstraction so that it can be used without knowledge of bytecode or the data format of the class file. Also, Javassist was designed to make it difficult to wrongly produce a class file rejected by the bytecode verifier of the JVM.

3.3 The Javassist API

We below present the overview of the Javassist API.

Reification and Reflection: The first step of the use of Javassist is to create a CtClass (compile-time class) object representing the bytecode of a class loaded into the JVM. This step is for reifying the class to make it accessible from a program. If `stream` is an InputStream for reading a class file (from a local disk, memory, a network, etc.), then:

```
CtClass c = new CtClass(stream);
```

creates a new CtClass object representing the bytecode of the class read from the class file, which contains enough symbolic information to reify the class. Also, the constructor of CtClass can receive a String class name instead of an InputStream. If a String class name is given, Javassist searches a class path and finds an InputStream for reading a class file.

One can call various methods on the CtClass object for introspecting and altering the class definition. Changes of the class definition are reflected on the bytecode represented by that object. To obtain the bytecode for loading the altered class into the JVM, method `toBytecode()` is called on that object:

```
byte[] bytecode = c.toBytecode();
```

Loading the obtained bytecode into the JVM is regarded as the step for reflecting the CtClass object on the base level. Javassist provides several other methods for this step. For example, method `compile()` writes bytecode to a given output stream such as a local file and a network. Method `load()` directly loads the class into the JVM with a class loader provided by Javassist. It returns a Class object representing the loaded class. Recall that Class is included in the Java reflection API while CtClass is in Javassist.

Note that Javassist does not provide any framework for specifying how and what classes are processed with Javassist. The programmer of the class loader has freedom with respect to this framework. For example, the class loader may process classes with Javassist only if they are specified by a configuration file read at the beginning. It may process them according to a *hard-coded* algorithm.

Table 1. Methods in CtClass for introspection

Method		Description
String	getName()	gets the class name
int	getModifiers()	gets the class modifiers such as public
boolean	isInterface()	determines whether this object represents a class or an interface
CtClass	getSuperclass()	gets the super class
CtClass[]	getInterfaces()	gets the interfaces
CtField[]	getDeclaredFields()	gets the fields declared in the class
CtMethod[]	getDeclaredConstructors()	gets the constructors declared in the class
CtMethod[]	getDeclaredMethods()	gets the methods declared in the class

Javassist allows a user class loader to define a new class from scratch without reading any class file. This is useful if a program needs to dynamically define a new class on demand. To do this, a CtClass object must be created as follows:

```
CtClass c2 = new CtNewClass();
```

The created object c2 represents an empty class that has no methods or fields although methods and fields can be added to the class later through the Javassist API shown below. If toBytecode() is called on this object, then it returns the bytecode corresponding to that empty class.

Introspection: Javassist provides several methods for introspecting the class represented by a CtClass object. This part of the Javassist API is compatible with the Java reflection API except that Javassist does not provide methods for creating an instance or invoking a method because these methods are meaningless at load time. Table 1 lists selected methods for introspection.

CtClass objects returned by getSuperclass() and getInterfaces() are constructed from class files found on a class path. They represent the original class definitions and thus accept only introspection but not alteration. To alter a class, another CtClass object must be explicitly created with the new operator. Modifications to this object have no effect on the CtClass object returned by getSuperclass() or getInterfaces(). For example, suppose that a class C inherits from a class S. If a CtClass object for S is created with new and a method m() is added to that object, this modification is not reflected on the object returned by getSuperclass() on a CtClass object for C. The class C inherits m() from S only if the CtClass object created with new is converted into bytecode and loaded into the JVM.

The information about fields and methods is provided by objects separate from the CtClass object; it is provided by CtField objects obtained by getDeclaredFields() and CtMethod objects obtained by getDeclaredMethods(), respectively. The information about a constructor is also provided by a CtMethod object. Table 2 lists methods in CtField and CtMethod for introspection.

Table 2. Methods in CtField and CtMethod for introspection

Method	in CtField	Description
`String`	`getName()`	gets the field name
`CtClass`	`getDeclaringClass()`	get the class declaring the field
`int`	`getModifiers()`	gets the field modifiers such as `public`
`CtClass`	`getType()`	get the field type
Method	in CtMethod	Description
`String`	`getName()`	gets the method name
`CtClass`	`getDeclaringClass()`	get the class declaring the method
`int`	`getModifiers()`	gets the method modifiers such as `public`
`CtClass[]`	`getParameterTypes()`	gets the types of the parameters
`CtClass[]`	`getExceptionTypes()`	gets the types of the exceptions that the method may throw
`boolean`	`isConstructor()`	returns `true` if the method is a constructor
`boolean`	`isClassInitializer()`	returns `true` if the method is a class initializer

Table 3. Methods for alteration

Method	in CtClass	Description
`void`	`bePublic()`	make the class `public`
`void`	`beAbstract()`	make the class `abstract`
`void`	`notFinal()`	remove the `final` modifier from the class
`void`	`setName(String name)`	change the class name
`void`	`setSuperclass(CtClass c)`	change the super class
`void`	`setInterfaces(CtClass[] i)`	change the interfaces
`void`	`addConstructor(...)`	add a new constructor
`void`	`addDefaultConstructor()`	add the default constructor
`void`	`addAbstractMethod(...)`	add a new `abstract` method
`void`	`addMethod(...)`	add a new method
`void`	`addWrapper(...)`	add a new wrapped method
`void`	`addField(...)`	add a new field
Method	in CtField	Description
`void`	`bePublic()`	make the field `public`
Method	in CtMethod	Description
`void`	`bePublic()`	make the method `public`
`void`	`instrument(...)`	modify a method body
`void`	`setBody(...)`	substitute a method body
`void`	`setWrapper(...)`	substitute a method body

Alteration: A difference between Javassist and the standard Java reflection API is that Javassist provides methods for altering class definitions. Several methods for alteration are defined in CtClass (Table 3). These methods are categorized into methods for changing class modifiers, methods for changing class hierarchy, and methods for adding a new member. They were carefully selected to satisfy our design goals.

Our design goals are three. (1) The first goal is to provide source-level abstraction for programmers. Javassist was designed so that programmers can use it without knowledge of the Java bytecode. (2) The second goal is to execute structural reflection as efficiently as possible. (3) The last goal is to help programs perform structural reflection in a safe manner in terms of types.

As for the first goal, the most significant design decision was how programmers specify a method body. Suppose that a new method is added to a class. If a sequence of bytecode is used for specifying the body of that method, the programmers would get great flexibility but have to learn details of bytecode. To achieve the first goal, Javassist allows to copy a method body from another existing method although this design decision restricts the flexibility of the added method. The copied bytecode sequence is adjusted to fit the destination method. For example, the bytecode for accessing a member through the `this` variable contains a symbolic reference to the type of `this`. This reference is replaced with one to the class declaring the destination method.

Despite the well-known quasi-equivalence between Java source code and bytecode, the correspondence between source-level and bytecode-level alterations are not straightforward. Hiding the gap between the two levels from programmers is also a part of the first goal.

For example, `setName()` renames a class but it also substitutes the new name for all occurrences of the old name in the definition of that class, including method signatures and bodies. Modifying a single constant-pool item never performs this substitution. If a constructor calls another constructor in the same class (if it executes `this()`), then the bytecode of the former constructor is modified since the bytecode contains a symbolic reference to the name of the class declaring the latter constructor. This reference must be modified to indicate the new name.

`setSuperclass()` performs similar substitution. If it is called, all occurrences of the old super class name is replaced with a new name and all constructors are modified so that they call a constructor in the new super class. However, there is an exception to this substitution. If the name of the original super class is java.lang.Object (the root of the class hierarchy), `setSuperclass()` does not perform the substitution except it modifies constructors. This is because java.lang.Object is often used for representing any class. For example, although `addElement()` in java.util.Vector takes a parameter of class java.lang.Object, which is the super class of java.util.Vector, this never means that `addElement()` takes an instance of the super class.

The second design goal is to reduce overheads due to class loading with Javassist. Since we will use Javassist for implementing a mobile-agent system, in which Javassist inserts security-check code into bytecode, Javassist must transform bytecode received through a network as efficiently as possible. Mobile agents frequently move among hosts and thus we cannot ignore the loading time of the bytecode implementing the mobile agents.

Our design decision on how programmers specify a method body was influenced by the second goal as well as the first one. Javassist does not use source code for specifying the body of an added method. If source code is used, it must be compiled *on the fly* when a class is loaded into the JVM. A naive implementation of this source-code approach would produce a complete class definition including the added method at source level and then compile it with a Java compiler such as `javac`. As we show later, however, this implementation implies serious performance penalties. To achieve practical efficiency, we need a special compiler that can quickly compile only a method body. We did not adopt the source-code approach because of limitations of our resources. Instead, Javassist allows to copy a pre-compiled method body from a class to another. This approach does not imply overheads due to source-code compilation at load time.

The third design goal is to prevent programs to wrongly produce a class including type incorrectness. To achieve this goal, Javassist allows only limited kinds of alteration of class definitions. In general, reflective systems should impose some restrictions on structural reflection so that programs do not falsely collapse themselves with reflection. Suppose that a reflective system allows to remove a field from a class at runtime. If there are already instances of that class, is it appropriate that the system simply discards the value of the removed field of those instances?

Since erroneous bytecode produced with Javassist is rejected by the bytecode verifier, it can never damage the JVM. However, restricting the reflective capability of Javassist is still necessary because it is often awkward to correct a program producing erroneous bytecode. For this reason, Javassist does not provide methods for removing a method or a field from a class because they cause type incorrectness if there is a method accessing the removed method or field. Javassist also imposes restrictions on the class passed to `setSuperclass()`, which is a method for changing a super class. The new super class must be a subclass of the original super class since there may be methods that implicitly cast an instance of that class to the original super class. Of course, the new super class must not be `final`. Furthermore, Javassist does not provide a method for changing the parameters of a method. Programmers are recommended to add a new method with the same name but with different parameters.

Adding a new member: Javassist provides methods for adding a new method to a class. To avoid the abstraction and performance problems mentioned above, `addMethod()` receives a `CtMethod` object, which specifies a method body. The signature of `addMethod()` is as shown below:

```
void addMethod(CtMethod m, String name, ClassMap map)
```

`name` specifies the name of the added method. The method body is copied from a given method `m`. Since a method body is copied from an existing compiled method, no source-code compilation is needed at load time or no raw bytecode is given to `addMethod()`. Programmers can describe a method body in Java and compile it in advance. Javassist reads the bytecode of the compiled method and adds it to another class. This improves execution performance of Javassist since a compiler is not run at load time.

When a method body is copied, some class names appearing in the body can be replaced according to a hash table `map`.[1] For example, programmers can declare a class XVector:

```
public class XVector extends java.util.Vector {
  public void add(X e) {
    super.addElement(e);
  }
}
```

and copy the method `add()` into a class StringVector:

```
CtMethod m = /* method add() in XVector */;
CtClass c = /* class StringVector */;
ClassMap map = new ClassMap();
map.put("X", "java.lang.String");
c.addMethod(m, "addString", map);
```

The class name java.lang.String is substituted for all occurrences of the class name X in `add()`. The added method is as follows:

```
public void addString(java.lang.String e) {
  super.addElement(e);
}
```

Javassist provides another method `addWrapper()` for adding a new method. It allows more generic description of a method body:

```
void addWrapper(int modifiers, CtClass returnType, String name,
                CtClass[] parameters, CtClass[] exceptions,
                CtMethod body, ConstParameter constParam)
```

The first five parameters specify the modifiers, the return type, the method name, the parameter types, and the exceptions that the method may throw. The body of the added method is copied from the method specified by `body`. No matter what the signature of the added method is, the method specified by `body` must have the following signature:

```
Object m(Object[] args, value-type constValue)
```

To fill the gap between this signature and the signature of the added method, `addWrapper()` implicitly wraps the copied method body in *glue* code, which constructs an array of actual parameters passed to the added method and assigns it to `args` before executing the copied method body. The glue code also sets `constValue` to a constant value specified by `constParam` passed to `addWrapper()`. In the current version of Javassist, an integer value or a `String`

[1] At least, `addMethod()` replaces all occurrences of the name of the class declaring the copied method. Even if that class name does not appear at source level, the corresponding bytecode may include references to it.

object can be specified for the constant value. For example, this constant value can be used to pass the name of the added method.

The value returned by the copied method body is an Object object. The glue code also converts it into a value of the type specified by `returnType`. Then it returns the converted value to the caller to the added method. If type conversion fails, then an exception is thrown. Although methods added by `addWrapper()` involve runtime overheads due to type conversion, a single method body can be used as a template of multiple methods receiving a different number of parameters. Examples of the use of `addWrapper()` are shown in Section 4.

Javassist also provides a method for adding a new field to a class:

```
void addField(int modifiers, CtClass type, String fieldname,
              String accessor, FieldInitializer init)
```

If `accessor` is not `null`, this method also adds an accessor method, which returns the value of the added field. The name of the accessor is specified by `accessor`. Moreover, the last parameter `init` specifies the initial value of the added field. The initial value is either one of parameters passed to a constructor, a newly created object, or the result of a call to a static method.

Altering a method body: Although Javassist does not allow to remove a method from a class, it provides methods for changing a method body. `setBody()` and `setWrapper()` in CtMethod substitute a given method body for an original body:

```
void setBody(CtMethod m, ClassMap map)
void setWrapper(CtMethod m, ConstParameter param)
```

They correspond to `addMethod()` and `addWrapper()` respectively. `setBody()` copies a method body from a given method `m`. Some class names appearing in the body are replaced with different names according to `map`. `setWrapper()` also copies a method body from `m` but it wraps the copied body in glue code. The signature of `m` must be:

`Object m(Object[] args,` *value-type* `constValue)`

Javassist also provides a method for modifying expressions in a method body. `instrument()` in CtMethod performs this modification:

```
void instrument(CodeConverter converter)
```

The parameter `converter` specifies how to instrument a method body. The CodeConverter object can perform various kinds of instrumentation. Table 4 lists methods provided by the current implementation of Javassist. They direct a CodeConverter object to replace a specific kind of expressions with *hooks*, which invoke static methods for executing the expressions in a customized manner. The idea of CodeConverter came from C++'s operator overloading. CodeConverter was

Table 4. Methods in CodeConverter

Method	Description
`void redirectFieldAccess()`	change a field-access expression to access a different field.
`void replaceNew()`	replace a `new` expression with a `static` method call.
`void replaceFieldRead()`	replace a field-read expression with a `static` method call.
`void replaceFieldWrite()`	replace a field-write expression with a `static` method call.

designed for safely altering the behavior of operators such as `new` and `.` (dot) independently of the context.

For example, expressions for instantiating a specific class can be replaced with expressions for calling a static method. Suppose that variables `xclass` and `yclass` represent class X and Y, respectively. Then a program:

```
CtMethod m = ... ;
CodeConverter conv = new CodeConverter();
conv.replaceNew(xclass, yclass, "create");
m.instrument(conv);
```

instruments the body of the method represented by the CtMethod object `m`. All expressions for instantiating the class X such as:

```
new X(3, 4);
```

are translated into expressions for calling a static method `create()` declared in the class Y:

```
Y.create(3, 4);
```

The parameters to the `new` expression are passed to the static method.

Reflective class loader: The class loader provided by Javassist allows a loaded program to control the class loading by that class loader. If a program is loaded by Javassist's class loader L and it includes a class C, then it can intercept the loading of C by L to self-reflectively modify the bytecode of C (Figure 1). For avoiding infinite recursion, while the loading of a class is intercepted, further interception is prohibited. The `load()` method in CtClass requires that a program is loaded by Javassist's class loader although the other methods work without Javassist's class loader.

Java's standard class loader never allows this self-reflective class loading for security reasons. If it is allowed, a program may change some `private` fields to `public` ones at load time for reading hidden values. Furthermore, in Java, if a program creates a class loader and loads a class C with that class loader, the loaded class is regarded as a different one from the class denoted by the name C appearing in that program. The latter class is loaded by the class loader that loaded the program.

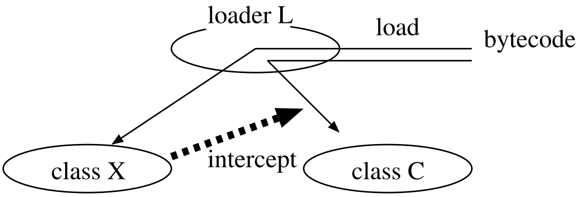


Fig. 1. Javassist's class loader allows self-reflective class loading

Using Javassist without a class loader: Javassist can be used without a user class loader. There are three kinds of usage of Javassist: with a user class loader, with a web server, and off line.

For security reasons, an applet is usually prohibited from using a user class loader. However, we can write an applet working with Javassist if we use a web server as a replacement of a user class loader. Since classes used in an applet are loaded from a web server into the JVM of a web browser, we can customize the web server so that it runs Javassist for processing the classes before sending them to the web browser. Javassist includes a simple web server written in Java as a basis for such customization. We can extend it to perform structural reflection with Javassist. The program of the customized web server would be as follows:

```
for (;;) {
  receive an http request from a web browser.
  CtClass c = new CtClass(the requested class);
  do structural reflection on c if needed.
  byte[] bytecode = c.toBytecode();
  send the bytecode to the web browser.
}
```

Before sending a requested class to a web browser, it performs structural reflection on the class according to the algorithm, for example, given as a configuration file.

Another usage of Javassist is "off line". We can perform structural reflection on a class and overwrite the original class file of that class with the bytecode obtained as the result. The altered class can be later loaded into the JVM without a user class loader. The following is an example of the off-line use of Javassist:

```
CtClass c = new CtClass("Rectangle");
do structural reflection on c if needed.
c.compile();     // writes bytecode on the original class file.
```

This program performs structural reflection on class Rectangle and overwrites the class file of that class with the bytecode obtained by `c.toBytecode()`.

4 Examples

This section shows three applications of Javassist. We illustrate that Javassist can be used to implement non-trivial alteration required by these applications despite the level of the abstraction.

4.1 Binary Code Adaptation

The mechanism of binary code adaptation (BCA) [13] automatically alters class definitions according to a file written by the users, called a delta file:

```
delta class implements Writable {
  rename Writable Printable;
  add public void print() { write(System.out); }
}
```

This delta file specifies adaptation that we mentioned in Section 2.

If Javassist is used, the implementor of BCA has only to write a parser of delta file and a user class loader performing adaptation with Javassist. For example, the parser translates the delta file shown above into the Java program shown below:

```
class Exemplar implements Printable {
  public void write(PrintStream s) { /* dummy */ }
  public void print() { write(System.out); }
}

class Adaptor {
  public void adapt(CtClass c) {
    CtMethod printM = /* method print() in Exemplar */;
    CtClass[] interfaces = c.getInterfaces();
    for (int i = 0; i < interfaces.length; ++i)
      if (interfaces[i].getName().equals("Writable")) {
        interfaces[i] = CtClass.forName("Printable");
        c.setInterfaces(interfaces);
        c.addMethod(printM, new ClassMap());
        return;
      }
  }
}
```

The class Exemplar is compiled together with Adapter in advance so that adapt() can obtain a CtMethod object representing print(). adapt() uses the reification and introspection API of Javassist for obtaining it. It first constructs a CtClass object representing Exemplar and then obtains the CtMethod object by getDeclaredMethods() in CtClass. The class file for Exemplar is automatically found by Javassist on the class path used for loading Adapter.

The user class loader calls adapt() in Adaptor whenever a class is loaded into the JVM. It creates a CtClass object representing the loaded class and calls adapt() with that object. The method adapt() performs adaptation if the loaded class implements Writable. Then the user class loader converts the CtClass object into bytecode and loads into the JVM.

Note that this implementation is more intuitive than the implementation with behavioral reflection. Moreover, it is simpler than the implementation without reflection since the implementor does not have to care about low-level bytecode transformation. If the users of BCA can directly write the classes Exemplar and Adaptor instead of a delta file, then the implementation would be much simpler since we do not need the parser of delta file.

4.2 Behavioral Reflection

Behavioral reflection enabled by MetaXa [16,11] and Kava [21] can be implemented with an approximately 750-line program (including comments) using Javassist. A key idea of their implementations is to insert *hooks* in a program when a class is loaded into the JVM. We below see an overview of a user class loader performing this insertion with Javassist.

Let a metaobject be an instance of MyMetaobject, which is a subclass of Metaobject:

```
public class MyMetaobject extends Metaobject {
  public Object trapMethodcall(String methodName, Object[] args) {
    /* called if a method call is intercepted. */ }
  public Object trapFieldRead(String fieldName) {
    /* called if the value of a field is read. */ }
  public void trapFieldWrite(String fieldName, Object value) {
    /* called if a field is set. */ }
}
```

If field accesses and method calls on an instance of C:

```
public class C {
  public int m(int x) { return x + f; }
  public int f;
}
```

are intercepted by the metaobject, then the user class loader alters the definition of the class C into the following:[2]

```
public class C implements Metalevel {
  public int m(int x) { /* notify a metaobject */ }
  public int f;
  private Metaobject _metaobject = new MyMetaobject(this);
  public Metaobject _getMetaobject() { return _metaobject; }
  public int orig_m(int x) { return x + f; }
  public static int read_f(Object target) {
      /* notify a metaobject */ }
  public static void write_f(Object target, int value) {
      /* notify a metaobject */ }
}
```

where the interface Metalevel declares the method _getMetaobject().

[2] For simplicity, this implementation ignores static members although extending the implementation for handling static members is possible within the ability of Javassist.

```
class Exemplar {
  private Metaobject _metaobject;

  public Object trap(Object[] args, String methodName) {
    return _metaobject.trapMethodcall(methodName, args);
  }

  public static Object trapRead(Object[] args, String name) {
    Metalevel target = (Metalevel)args[0];
    return target._getMetaobject().trapFieldRead(name);
  }

  public static Object trapWrite(Object[] args, String name) {
    Metalevel target = (Metalevel)args[0];
    Object value = args[1];
    target._getMetaobject().trapFieldWrite(name, value);
  }
}
```

Fig. 2. Class Exemplar

This alteration can be performed within the ability of Javassist. The interface Metalevel is added by `setInterfaces()` in CtClass. The field `_metaobject` and the accessor `_getMetaobject()` are added by `addField()` in CtClass.

For intercepting method calls, the user class loader first makes a copy of every method in C by calling `addMethod()` in CtClass. For example, it adds `orig_m()`[3] as a copy of `m()`. Then it replaces the body of every method in C with a copy of the body of the method `trap()` in Exemplar (see Figure 2). This modification is performed by `setWrapper()` in CtMethod. The gap between the signatures of `m()` and `trap()` is filled by `setWrapper()`. The substituted method body notifies a metaobject of interception. The first parameter `args` is a list of actual parameters and the second one `name` is the name of the copy of the original method such as `"orig_m"`. These two parameters are used for the metaobject to invoke the original method through the Java reflection API.

For intercepting field accesses, the user class loader instruments the bodies of methods in all classes. All accesses to a field `f` in C are translated into calls to a `static` method `read_f()` or `write_f()`. This instrumentation is performed by `instrument()` in CtMethod and `replaceFieldRead()` and `replaceFieldWrite()` in CodeConverter. The methods `read_f()` and `write_f()` notify a metaobject of the accesses. They are added by `addWrapper()` in CtClass as copies of `trapRead()` and `trapWrite()` in Exemplar. The gap between the signatures of `read_f()` (or `write_f()`) and `trapRead()` (or `trapWrite()`) is filled by `addWrapper()`. For example, actual parameters to `read_f()` are converted into the first parameter `args` to `trapRead()`. The second parameter `name` to `trapRead()` is the name of the accessed field such as `"f"`.

[3] If a method name is overloaded, a copy of each method must be given a different name such as `orig_m1()`, `orig_m2()`, ...

4.3 Remote Method Invocation

Generating stub code for remote method invocation is another application of Javassist. A Java program cannot directly call a method on a remote object on a different computer. It needs the Java RMI tools generating stub code, which translates a method call into lower-level network data transfer such as TCP/IP communication. However, the Java RMI tools are compile-time ones; a program must be processed by the RMI compiler, which generates and saves stub code on a local disk. Also, a program using the Java RMI must be subject to a protocol (i.e. API) specified by the Java RMI.

Javassist allows programmers to develop their own version of the RMI tools, which specify a customized protocol and produce stub code at either compile-time or even runtime. Suppose that an applet needs to call a method on a Counter object on a web server written in Java. For remote method invocation, the applet needs stub code defining a proxy object of the Counter object, which has the same set of methods as the Counter object. If the Counter object has a method `setCount()`, the proxy object also has a method `setCount()` with the same signature. However, the method on the proxy object serializes given parameters and sends them to the web server, where `setCount()` is invoked on the Counter object with the received parameters.

This stub code can be generated at runtime with Javassist at the server side and it can be sent on demand to the applet side. The applet programmer can easily write the applet without concern about low-level network programming. The stub code for accessing the Counter object is as follows:

```
public class ProxyCounter {
  private RmiStream rmi;
  public ProxyCounter(int objectRef) {
    rmi = new RmiStream(objectRef);
  }
  public int setCount(int value) { /* remote method invocation */ }
}
```

An instance of ProxyCounter is a proxy object. An RmiStream object handles low-level network communication. The class RmiStream is provided by a runtime support library.

ProxyCounter can be defined within the confines of Javassist. The field `rmi` is added by `addField()` in CtClass and the initialization of `rmi` in a constructor can be specified by a FieldInitializer object passed to `addField()`.

The method `setCount()` is added by `addWrapper()` in CtClass as a copy of the method `invoke()` in Exemplar shown below:

```
class Exemplar {
  private RmiStream rmi;
  Object invoke(Object[] args, String methodName) {
    return rmi.rpc(methodName, args);
  }
}
```

The gap between the signatures of `setCount()` and `invoke()` is filled by `addWrapper()`. If `setCount()` is called, the actual parameter `value` is converted into an array of Object and assigned to `args`. `methodName` is set to a method name `"setCount"`[4]. Then `rpc()` is called on the RmiStream object for serializing the given parameters and sends them to the web server. Note that the parameters can be serialized within the ability of the standard Java if they are converted into an array of Object.

Stub code generation is another example, which is not straightforward to implement with behavioral reflection. In a typical implementation with behavioral reflection, a proxy object is an instance of the class Counter although all method calls on the proxy object are intercepted by a metaobject and forwarded to a remote object; the class ProxyCounter is not produced. Therefore, if the proxy object is created, a constructor declared in Counter is called and may cause fatal side-effects since the class Counter is defined as a class at the server side but the proxy object is not at that side.

5 Related Work

Reflection in Java: MetaXa [16,11] and Kava [21] enable behavioral reflection in Java whereas Javassist enables structural reflection. They are suitable for implementing different kinds of language extensions. However, Javassist indirectly covers applications of MetaXa and Kava since a class loader providing functionality equivalent to MetaXa and Kava can be implemented with Javassist as we showed in Section 4.2.

Although Kava performs bytecode transformation of class files before the JVM loads them as Javassist does, they only insert hooks for interception in bytecode but do not run metaobjects at that time. They enable reflection at runtime and their ability is not structural reflection but the restricted behavioral reflection.

The Java reflection API was recently extended in the JDK 1.3 beta to partially enable behavioral reflection [19]. The new API allows a program to dynamically define a proxy class implementing given interfaces. An instance of this proxy class delegates all method invocations to another object through a type-independent interface.

Javassist is not the first system enabling structural reflection in Java. For example, Kirby et al proposed a system enabling structural reflection (they called it linguistic reflection) in Java although their system only allows to dynamically define a new class but not to alter a given class definition at load time [15]. With their system, a Java program can produce a source file of a new class, compile it with an external compiler such as `javac`, and load the compiled class with a user class loader. They reported that their system could be used for defining a class optimized for a given runtime condition.

[4] If a method name is overloaded, this should be `setCount1`, `setCount2`, ... for distinction.

Compile-time metaobject protocol: The compile-time metaobject protocol [3] is another architecture enabling structural reflection without modifying an existing runtime system. OpenJava [20] is a Java implementation of this architecture. As Javassist does, it restricts structural reflection within the time before a class is loaded into the JVM although it was designed mainly for off-line use at compile time. However, OpenJava is source-code basis although Javassist is bytecode basis; OpenJava reads source code for creating an object representing a class, a method, or a field. Alteration to the object is translated into corresponding transformation of the source code. The bytecode for the altered class is obtained by compiling the modified source code. Since OpenJava is source-code basis, it can deal with syntax extensions within a framework of structural reflection. For example, one can extend the syntax of class declaration and make it possible to add an annotation to a class declaration.

On the other hand, the source-code basis means that OpenJava needs the source file of every processed class whereas Javassist needs only a class file (compiled binary). This is a disadvantage because source files are not always available if the class is provided by a third party. OpenJava also involves a performance overhead due to handling source code; the source file of every class must be parsed for reification and compiled for reflection. Although this overhead is compensation for the capability for fine-grained transformation of source code (including syntax extension), it is not negligible if OpenJava is used by a class loader for altering a loaded class. Some kinds of applications such as a mobile agent system do not need fine-grained transformation but fast class loading.

Although the implementations of OpenJava or Javassist have not been tuned up, the performance difference between OpenJava and Javassist is notable with respect to reification and reflection. If a class loader can be implemented with either OpenJava or Javassist, Javassist achieves shorter loading time. To show this performance difference, we compared Javassist and OpenJava with two small applications. We implemented BCA [5] and behavioral reflection presented in Section 4 with both Javassist and OpenJava and we measured the time needed for altering a given class with each implementation. For fair comparison, the implementations with Javassist write modified class files back on a local disk.

Table 5 lists the results. The execution time is the average of five continuous repetitions, which do not include the first repetition. Since a program is gradually loaded into the JVM during the first repetition, the first one is tremendously slow. For compiling a modified source file, OpenJava uses a compiler provided by the Sun JDK for Solaris. However, it never uses the `javac` command since it starts the compiler in a separate process; instead, it directly runs the compiler (`sun.tools.javac`) on the same JVM.

Although the sizes of the programs implementing the applications are almost equal between Javassist and OpenJava, Javassist processed a class more than ten times faster than OpenJava. Note that the execution time by Javassist is shorter than the time needed only for compiling a modified source file. This is because

[5] Of course, the implementation of BCA with OpenJava does not modify a class file in binary form. It emulates equivalent adaptation at source-code level.

Table 5. Performance comparison between Javassist and OpenJava

		execution time (msec)	program size (lines)	original source (lines)	 class file (bytes)	modified class file (bytes)
BCA	Javassist	42	26	24	372	551
	OpenJava	543 (172†)	17	24		548
Reflection	Javassist	142	205	35	946	3932
	OpenJava	4108 (302†)	247	35		2244

Sun JDK 1.2.2 (HotSpot VM 1.0.1), UltraSPARC II 300MHz
†compilation time by `sun.tools.javac` (Java compiler).

Javassist can move compilation penalties to an earlier stage. Even a method body is not compiled while Javassist is running; it is pre-compiled in advance and the resulting bytecode is directly copied to a target class at run time.

Bytecode translators: Bytecode translators such as JOIE [5] and the JavaClass API [7] provide a functionality similar to Javassist. They enable a Java program to alter a class definition at load time. However, they are toolkits for directly dealing with bytecode, that is, the raw data structure of a class file. For example, classes included in JOIE are ClassInfo, Code, and Instruction. They show that JOIE was designed for experienced programmers who have a deep understanding of the Java bytecode and want to implement complex transformation. On the other hand, Javassist was designed to be easy to use; it does not require programmers to have knowledge of the Java bytecode but instead it provides source-level abstraction for manipulating bytecode in a relatively safe manner. Although a range of instrumentation of a method body is restricted, we showed that Javassist can be used to implement non-trivial applications. Javassist can be regarded as a front end for easily and safely using a bytecode translator like JOIE; it is not a replacement of the bytecode translators.

Using bytecode instrumentation for implementing a reflective facility is a known technique in Smalltalk [1]. A uniqueness of Javassist against this is the design of the API providing source-level abstraction. The Javassist API was carefully designed to avoid wrongly producing a class definition containing type incorrectness.

Others: OpenJIT [18] is a just-in-time compiler that allows a Java program to control how bytecode are compiled into native code. It provides better flexibility than Javassist with respect to instrumenting a method body while OpenJIT does not allow to add a new method or field to a class. However, using OpenJIT is more difficult than using Javassist because OpenJIT requires programmers to have knowledge of both the Java bytecode and native code. Although OpenJIT can be used without knowledge of the Java bytecode if programmers use a mechanism of OpenJIT for translating bytecode into a parse tree of an equivalent Java program, overheads due to that translation has not been reported.

The idea of enabling reflection only at load time for avoiding performance problems is found in the CLOS MOP [14]. For example, the CLOS MOP allows a program to alter the algorithm of determining the super classes of a given class but the super classes are statically determined when the class is loaded; the program cannot dynamically change the super classes at runtime.

Some readers may think that Javassist is very similar to BCA. However, Javassist was designed for a wider range of applications than BCA, which is specialized for on-line class adaptation. BCA only allows to modify a given class but not to dynamically define a new class from scratch. On the other hand, BCA allows programmers to describe the algorithm of adaptation in declarative form.

6 Conclusion

This paper presented Javassist, which is an extension to the Java reflection API. Unlike other extensions, it enables structural reflection in Java; it allows a program to alter a given class definition and to dynamically define a new class. A number of language extensions are more easily implemented with structural reflection than with behavioral reflection.

For avoiding portability and performance problems, the design of Javassist is based on our new architecture for structural reflection. Javassist performs structural reflection by instrumenting bytecode of a loaded class. Therefore, it can be used with a standard JVM and compiler although structural reflection is allowed only before a class is loaded into the JVM, that is, at load time. Since a standard JVM is used, the classes processed by Javassist are subject to the bytecode verifier and the SecurityManager of Java. Javassist never breaks security guarantees given by Java.

The followings are important features of Javassist:

- Javassist is portable. It is implemented in only Java without native methods and it runs with a standard JVM. It does not need a platform-dependent class library. Portability is significant in Java programming.
- Javassist provides source-level abstraction for manipulating bytecode in a safe manner while bytecode translators, such as JOIE [5] and the JavaClass API [7], provide no higher-level abstraction. The users of Javassist do not have to have a deep understanding of the Java bytecode or to be careful for avoiding wrongly making an invalid class rejected by the bytecode verifier.
- Javassist never needs source code whereas OpenJava [20], which is another system for structural reflection with source-level abstraction, does. Since OpenJava performs structural reflection by transforming source code, it must parse and compile source code for reifying and reflecting a class. Thus a class loader using Javassist can load a class faster than one using OpenJava. However, OpenJava enables fine-grained manipulation of class definitions so that the resulting definitions may be smaller and more efficient than ones by Javassist.

The architecture that we designed for Javassist can be applied to other object-oriented languages if a compiled binary program includes enough symbolic information to construct a class object. However, the API must be individually designed for each language so that it allows a program to alter class definitions in a safe manner with respect to the semantics of that language.

Acknowledgment. The author thanks Michiaki Tatsubori, who wrote programs using OpenJava for the experiment in Section 5. He also thanks Hidehiko Masuhara for his comments on an early draft of this paper, and the anonymous reviewers for their helpful comments.

References

1. Brant, J., B. Foote, R. E. Johnson, and D. Roberts, “Wrappers to the Rescure,” in *ECOOP’98 - Object Oriented Programming*, LNCS 1445, pp. 396–417, Springer, 1998.
2. Braux, M. and J. Noyé, “Towards Partially Evaluating Reflection in Java,” in *Proc. of Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM’00)*, SIGPLAN Notices vol. 34, no. 11, pp. 2–11, ACM, 1999.
3. Chiba, S., “A Metaobject Protocol for C++,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices vol. 30, no. 10, pp. 285–299, ACM, 1995.
4. Chiba, S. and T. Masuda, “Designing an Extensible Distributed Language with a Meta-Level Architecture,” in *Proc. of the 7th European Conference on Object-Oriented Programming*, LNCS 707, pp. 482–501, Springer-Verlag, 1993.
5. Cohen, G. A., J. S. Chase, and D. L. Kaminsky, “Automatic Program Transformation with JOIE,” in *USENIX Annual Technical Conference ’98*, 1998.
6. Cointe, P., “Metaclasses are first class: The ObjVlisp model,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pp. 156–167, 1987.
7. Dahm, M., “Byte Code Engineering with the JavaClass API,” Techincal Report B-17-98, Institut für Informatik, Freie Universität Berlin, January 1999.
8. Ellis, M. and B. Stroustrup, eds., *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.
9. Fabre, J. C. and T. Pérennou, “A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach,” *IEEE Transactions on Computers*, vol. 47, no. 1, pp. 78–95, 1998.
10. Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.
11. Golm, M. and J. Kleinöder, “Jumping to the Meta Level, Behavioral Reflection Can Be Fast and Flexible,” in *Proc. of Reflection ’99*, LNCS 1616, pp. 22–39, Springer, 1999.
12. Hölzle, U. and D. Ungar, “A Third Generation Self Implementation: Reconciling Responsiveness with Performance,” in *Proc. of ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications*, SIGPLAN Notices vol. 29, no. 10, pp. 229–243, 1994.
13. Keller, R. and U. Hölzle, “Binary Component Adaptation,” in *ECOOP’98 - Object Oriented Programming*, LNCS 1445, pp. 307–329, Springer, 1998.

14. Kiczales, G., J. des Rivières, and D. G. Bobrow, *The Art of the Metaobject Protocol.* The MIT Press, 1991.
15. Kirby, G., R. Morrison, and D. Stemple, "Linguistic Reflection in Java," *Software – Practice and Experience*, vol. 28, no. 10, pp. 1045–1077, 1998.
16. Kleinöder, J. and M. Golm, "MetaJava: An Efficient Run-Time Meta Architecture for Java," in *Proc. of the International Workshop on Object Orientation in Operating Systems (IWOOS'96)*, IEEE, 1996.
17. Masuhara, H. and A. Yonezawa, "Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Languages," in *ECOOP'98 - Object Oriented Programming*, LNCS 1445, pp. 418–439, Springer, 1998.
18. Ogawa, H., K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and F. Kimura, "OpenJIT : An Open-Ended, Reflective JIT Compiler Framework for Java," in *Proc. of ECOOP'2000*, Springer Verlag, 2000. To appear.
19. Sun Microsystems, "Java^{TM} 2 SDK Documentation." version 1.3 (beta release), 1999.
20. Tatsubori, M., S. Chiba, M.-O. Killijian, and K. Itano, "OpenJava: A Class-based Macro System for Java," in *Reflection and Software Engineering* (W. Cazzola, R. J. Stroud, and F. Tisato, eds.), LNCS 1826, Springer Verlag, 2000.
21. Welch, I. and R. Stroud, "From Dalang to Kava — The Evolution of a Reflective Java Extension," in *Proc. of Reflection '99*, LNCS 1616, pp. 2–21, Springer, 1999.
22. Wu, Z., "Reflective Java and A Reflective-Component-Based Transaction Architecture," in *Proc. of OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (J.-C. Fabre and S. Chiba, eds.), 1998.

Runtime Support for Type-Safe Dynamic Java Classes

Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes

Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{malabarb, pandey, gragg, barr, barnes}@cs.ucdavis.edu
http://pdclab.cs.ucdavis.edu/

Abstract. Modern software must evolve in response to changing conditions. In the most widely used programming environments, code is static and cannot change at runtime. This poses problems for applications that have limited down-time. More support is needed for dynamic evolution. In this paper we present an approach for supporting dynamic evolution of Java programs. In this approach, Java programs can evolve by changing their components, namely classes, during their execution. Changes in a class lead to changes in its instances, thereby allowing evolution of both code and state. The approach promotes compatibility with existing Java applications, and maintains the security and type safety controls imposed by Java's dynamic linking mechanism. Experimental analyses of our implementation indicate that the implementation imposes a moderate performance penalty relative to the unmodified virtual machine.

1 Introduction

Software systems must change over time. Changing business practices, the relentless advance of technology, and the demands of end users drive this evolution. The functionality required of applications inevitably changes in response to these factors. Consequently, in order to remain viable, applications must evolve to meet new requirements. Software component evolution is a major focus of effort in software engineering [27,38].

The vast majority of commercial software is written in a few imperative languages, such as C++ or Java [3]. For these languages, software evolution is generally a slow, static process. Most of us are familiar with the process of waiting for the latest version of our favorite program to come out, stopping work to install the new version over the old one, then cleaning up the resultant mess of incompatible document formats and lost settings. The fact that a running program cannot be changed drives this cycle. Since any update requires stopping a program and overwriting all or part of it, incremental updates are often impractical, and major updates problematic. For a large class of critical applications, such as business transaction systems, telephone switching systems and emergency response systems, the interruption poses an unacceptable loss of availability.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 337–361, 2000.
© Springer-Verlag Berlin Heidelberg 2000

What is needed, then, is more support for applications that evolve *during execution*. In addition to supporting true evolution of software systems, dynamic evolution provides several other benefits:

- *Software distribution and management*: Dynamic evolution has applications in software distribution and management. Consider a distributed system in which changes in all active applications are either pulled or pushed from software servers to the active applications. While several applications, for instance NetscapeNavigator, MicrosoftInternet Explorer and RealAudioRealPlayer, currently support such application-specific updates, most use static updates for modifying applications.
- *Runtime optimization*: Often, specific properties of systems are best determined at runtime. For example, many applications can be highly optimized *if* some information about the input is known during development. However, these same optimizations result in specialized code restricted to a smaller input domain. If code can be modified at runtime, a program can accept a wider range of data, yet load and use methods optimized for the current data set.
- *Dynamic policy specification*: Dynamic evolution can be very useful in any application whose behavior is driven by a set of policies, for instance security policies. For example, dynamic security policies can be implemented using total mediation, without modifying code at runtime. This method requires a security check at every access of every resource [36]; due to the high performance cost, it is not widely used. Systems that employ total mediation implement dynamic policies by using general, static code to interpret dynamic data structures – a computationally expensive process. Dynamic evolution allows designers to move logic from interpreted data structures into directly executed code. This provides the efficiency of code-driven security enforcement [35] without sacrificing flexibility.

In this paper, we present an approach for dynamic evolution of Java programs. While Java [3] provides several mechanisms, such as inheritance, interfaces and dynamic linking, for program extensibility, it does not support true dynamic evolution, in which both the code and state of a program can evolve gracefully. In our approach, Java programs can evolve by changing their components, namely classes, during execution.

Java classes can be considered to have a life cycle with three discrete states: unloaded, or *static*, *loaded*, and *active*. A static class exists only in storage; it has not been loaded into the Java virtual machine. A loaded class has been loaded and possibly linked. Finally, an active class has live instances and/or methods running. We are concerned with changing active classes; a *dynamic class* can change while active.

We wished to preserve the syntax and semantics of the target language. Doing so ensures compatibility with existing code, and provides greater ease of use as developers do not need to learn new language constructs. This constraint requires that we preserve the type safety characteristics of a program throughout its execution. Type safety encourages the development of safer, more disciplined code. In a dynamic system, type safety can restrict wild, unsound changes, alleviating the dangers inherent in changing code. Further, many of Java's security mechanisms, for instance separation of user and system name spaces and protection of private data, depend on the type-safe properties of Java programs. Therefore, we impose the restriction that all changes in a program preserve the type safety properties of the program. Section 2.1 presents our formal model, and defines valid class change. Using the formal model, we show that a valid class change preserves type safety. Further, in order to provide a convenient, backward-compatible interface, and to support changes in any Java class, we extended the Java class loader [29]. The new dynamic class loader allows a program to define a class multiple times. The dynamic class loader implements changes in a class, and any resulting changes in its instances, in an executing program. We describe the dynamic class loader in detail in Section 2.2.

Support for dynamic evolution, however, raises additional security issues, as malicious applets may use the dynamic class mechanism to modify the classes that enforce specific security policies of a host. Therefore, the dynamic class loader implements a security model that ensures that Java programs can dynamically modify only those resources to which they are authorized. We enforce this policy using name space separation and resource access control. We discuss the security model in Section 2.3.

We have implemented support for dynamic classes by modifying Sun's Java virtual machine (JDK 1.2). Dynamic classes can be implemented in several ways: by changing the language, through library-based support, or by modifying the virtual machine. As stated above, we did not wish to change the language. Library-based support proved to be too awkward and inefficient for our requirements. Thus, we chose to directly modify the virtual machine. Section 3 describes our implementation in detail.

We performed several experiments to measure the performance characteristics of our implementation. The experiments show that dynamic classes add about 6-10% of overhead to Sun's JVM. Further, the cost of updating classes is moderate. Section 4 presents these results, as well as further analysis with regard to alternative methods and related work.

2 Dynamic Classes

In this section, we formally describe the concept of dynamic classes. We begin by presenting a formal model of classes, objects, and inheritance in Java. We consider the potential effects of introducing dynamic classes into a running application. We then use our model to define type-safe dynamic classes. We discuss how best to support dynamic classes while addressing type safety and security issues, and describe our design. In general, we have made conservative design choices, emphasizing compatibility with existing Java code, minimizing performance penalties, and maintaining type safety and overall system security.

2.1 Formal Model

We begin by formalizing the notion of classes, interfaces, inheritance, composition, and dependency among classes in Java. In doing so, we build upon the formal Java type model developed in [9], which includes type widening [8,39].

Types, classes and objects: A type T denotes a set of objects. T is bound to a definition that describes the contents of the objects and operations that act on them. Specifically, this definition consists of T's *interface* and its *implementation.*

Definition 1. *Proof ((Type interface ($\mathfrak{i}(T)$)). The interface of a type T, $\mathfrak{i}(T)$, is a set of public data fields and methods.* □

Definition 2. *Proof ((Type implementation ($\mathfrak{b}(T)$)). The implementation, or body, of a type T, $\mathfrak{b}(T)$, is a set of private data fields and a set of method bodies.* □

Definition 3. *Proof ((Interface)). The Java interface construct describes, but does not implement, a type. An interface contains no data fields.* □

Definition 4. *Proof ((Class)). A class C is defined by the tuple $<\mathfrak{i}(C), \mathfrak{b}(C)>$, where $\mathfrak{b}(C)$ contains implementations for all methods declared in $\mathfrak{i}(C)$. Java supports abstract classes, which provide only a partial implementation.* □

Definition 5. *Proof ((Implements ($\geq_I$))). The relation $C \geq_I I$ is true if C implements the interface I: $\mathfrak{i}(I) \subseteq \mathfrak{i}(C)$.* □

Definition 6. *Proof ((Program)). A program is a set of classes.* □

A Java class C_1 *depends* on another class C_2 if $\mathfrak{b}(C_1)$ contains references to C_2. The references may include method invocations, field accesses and inheritance. Any change to C_2 may mean that C_1 must change as well [6].

Definition 7. *Proof ((Dependency ($\propto$))). The relation $C_1 \propto C_2$ is true if C_1 depends on C_2. Transitivity applies, denoted by $\overset{*}{\propto}$. The dependency relationship applies to specific methods or fields as well. For instance, $C_1.M \propto C_2.N$ is true if $C_1.M$ invokes $C_2.N$.* □

Definition 8. *Proof ((Composition ($\oplus$, $\ominus$))). $C_1 \oplus C_2$ denotes the union of C_1 and C_2, where C_1 and C_2 are two sets of methods and data. $C_1 \ominus C_2$ denotes their difference; the methods and fields that are defined in C_1, but not in C_2. These operators provide an abstraction for the Java inheritance mechanism. Thus, the Java composition semantics of scoping and overloading are implicit in the definitions of $\oplus$ and $\ominus$.* □

We do not define $\oplus$ and $\ominus$ precisely because our focus in this paper is more on examining the effects of dynamic classes.

Definition 9. *Proof ((Inheritance($\sqsubseteq$))). The relation $C \sqsubseteq C_S$ is true if C directly extends C_S. Inheritance affects the composition of a class. $\mathfrak{b}(C)$ contains the implementations of all of C's superclasses, and $\mathfrak{i}(I)$ contains their interfaces. Stated formally:*

$$\forall\ T : C \overset{*}{\sqsubseteq} T : \mathfrak{b}(T) \subseteq \mathfrak{b}(C)$$

$$\forall\ T : C \overset{*}{\sqsubseteq} T : \mathfrak{i}(T) \subseteq \mathfrak{i}(C)$$

Transitivity applies, denoted by $\overset{*}{\sqsubseteq}$. Java does not permit recursive inheritance. Thus, $C_1 \overset{*}{\sqsubseteq} C_2 \implies \neg\ (C_2 \overset{*}{\sqsubseteq} C_1)$. Finally, $C_1 \sqsubseteq C_2 \implies\ \ C_1 \propto C_2$. Inheritance can apply to both classes and interfaces. Let $\sqsubseteq_C$ specify class extension, and $\sqsubseteq_I$ specify interface extension. $\sqsubseteq$ can refer to either case. □

Definition 10. *Proof ((Defines: class ($\geq_C$))). The relation $C_{def} \geq_C C$ is true if C_{def} is the class definition bound to the name C; C_{def} defines C. A $\geq_C$ relationship is not necessarily permanent, but it is singular; $C_{def} \geq_C C \implies \forall$ C_i: $C_i \neq C_{def}$, $\neg$ ($C_i \geq_C C$). This restriction preserves Java name semantics — a name should only be bound to one value.* □

Definition 11. *Proof ((Instantiation ($\leq$))). The relation $O\ \leq T$ is true if the object O is an instance of the class or interface T. $O \leq C \wedge C \overset{*}{\sqsubseteq} D \implies O \leq D$. Likewise, $O \leq C \wedge C \geq_I I \implies O \leq I$.* □

Definition 12. *Proof ((Defines: object ($\geq_O$))). The relation $C_{def} \geq_O O$ is true iff $C_{def} \geq C \wedge O \leq C$. As with $\geq_C$, $\geq_O$ is not necessarily permanent, and is singular.* □

Note that $\leq$ is the transverse of $\geq_O$.

Type safety issues: Changing a class C can have a serious impact on type safety. The interface and/or implementation may be affected. Methods can be added, deleted, or modified, and data fields may be added or deleted. Furthermore, the type itself can change. Adding or removing superclasses or interfaces effectively changes the set of types that an instance of C can be cast or assigned to, with potential effects on any variables bound to such an object. Type violations caused by dynamic changes in class definitions fall into two categories: *static* type violations and *dynamic* type violations. The design and implementation of Java contain mechanisms to prevent either from occuring in a static program. Our system must also prevent them from occuring in a dynamic program, due to class changes.

Here we define static and dynamic type violations, and describe how both the standard JVM and our model prevent these violations and ensure type safety. In doing so, we use the notion of the *type set* of a class C, which is the set of all classes and interfaces to which an instance of C can be cast. The type set contains C itself, all classes from which it inherits, and all interfaces that C or one of its superclasses implements.

Definition 13. *Proof ((Type set ($\tau(C)$))). Let I_C be the set of all interfaces i such that $C \geq_I i$. Let C_S be C's superclass; $C \sqsubseteq C_S$. Then, $\tau(C) \equiv \{C\} \cup I_C \cup \tau(C_S)$.*
From the definition of instantiation, $O \leq C \implies \forall\, T\colon T \in \tau(C)\colon O \leq T$. □

A static type violation is an invalid field or method reference. For example, if a method in class C_1 references the field $C_2.X$, and C_2 does not contain a field called X, the reference to X is invalid. This type of violation can be detected statically by examining the source program. The Java compiler and dynamic linker detect static type violations in source code. This mechanism cannot prevent static type violations caused by dynamic class changes. For instance, a class D may invoke a method, say `C.foo()`, of a class C. Now, any dynamic changes in `C` that removes `foo` will cause invalid references to `foo`.

A dynamic type violation occurs when some event results in a reference being bound to an object of an incompatible type. For example, let O be an instance of C. C does *not* implement the interface I. If O is bound to a variable i of type I, a dynamic type violation results. This type violation cannot be detected statically, since it depends on O. The JVM performs dynamic type checking during operations such as assignment and type casting. If an operation might result in a dynamic type violation, the JVM throws an exception. This type of checking does not always catch dynamic type violations caused by class change, since an assignment might have occurred prior to the class change. For instance, assume that C implements interface I. Let O be an instance of C. Some other object has a reference to O, via i, of type I. If C changes such that it no longer implements I, i's reference to O becomes invalid, since O's type has changed. In this example, i has already been assigned a value. If O's type changes, then any subsequent access to i might cause an error. The only way to prevent such an error would be to type check every object reference instruction, which the JVM currently does not do.

We now show how dynamic type violations can occur when classes are changed dynamically. Assume that class C implements the interface I. Thus, $\tau(\mathrm{C}) \equiv \{C, I, Object\}$. We first assign an object of type C to a variable of type I, a legal action. We then modify C such that it no longer implements I; $\tau(\mathrm{C}) \equiv \{C, Object\}$. The reference `i.foo()` causes an error, because the object bound to `i` is no longer of type `I`.

We can now define type safety formally:

Definition 14. *Proof ((Type safety)). A class C is type-safe if it contains neither static type violations nor dynamic type violations that cannot be detected by the JVM's runtime type checking. A program P is type-safe if all of its component classes are type-safe.* □

There are two approaches to ensuring type safety during class changes. We could place no constraints on how classes can change, and type check every object reference and method invocation instruction. Or, reduce the necessity for extra runtime type checking by placing constraints on class changes. Various definitions of a valid class change are possible, depending on the approach used.

We have defined a valid class change as one that cannot cause type violations, either static or dynamic.

We chose this approach for two reasons. First, we wished to preserve the type semantics of the Java language. A valid Java program, P, does not contain these type violations. Second, efficiency – type checking all method and object references requires significant CPU time. Our model requires only static checking before a class is modified. No extra runtime type checking is necessary.

Formally, we define the semantics of class change to prevent static and dynamic type violations as follows:

Notation: Let $\underline{C}$ denote the definition bound to class C before a change.
Notation: Let $\overline{C}$ denote the definition bound to class C after a change.
Notation: Let ΔC denote the changes made between $\underline{C}$ and $\overline{C}$; $\Delta C \equiv (\underline{C} \ominus \overline{C}) \oplus (\overline{C} \ominus \underline{C})$.

Definition 15. *Proof ((Dynamic class change ($\mapsto$))). The operation $\underline{C} \mapsto \overline{C}$ describes a change to C's definition, and is valid if and only if the following two conditions hold true:*

1. *No class defined in P, where P is the enclosing program, depends on fields or methods being removed from C.*
   $\forall\ C_D \in P : \neg\ (C_D \overset{*}{\propto} (\underline{C} \ominus \overline{C}))$
2. *An element of C's type set cannot be removed if other classes depend on it.*
   $\forall T : T \in \tau(\underline{C} \ominus \overline{C}) : \neg(\exists\ C_D : C_D \neq C \wedge C_D \in P : C_D \propto T)$.

Under these conditions, $\mathfrak{b}(C)$ may be changed in any way. Methods and data may be added to $\mathfrak{i}(C)$, and removed if doing so does not cause type violations. C's superclass may be changed, and abstract interfaces added or removed as long as types with dependents are not removed from C's type set. Further, $\underline{C} \mapsto \overline{C}$ has the following effects on C subclasses and instances:

1. *The change in C's definition is reflected in all subclasses.*
   $\forall\ C_D\colon C_D \sqsubseteq C,\ \underline{C_D} \mapsto \overline{C_D}$.
   By the definition of inheritance, $\Delta C_D \equiv \Delta C$.
2. *All instances of C change to match the new defintion.*
   $\forall\ O\colon O \leq C,\ \underline{O} \mapsto \overline{O}$, *where* $\underline{C} \geq_O \underline{O}$ *and* $\overline{C} \geq_O \overline{O}$. *See Section 3.2 for more information about this requirement.*

□

Note that $C_D \overset{*}{\propto} C$ does *not* mean that C_D must change if C does. If C_D depends on C via method invocation, field access, or aggregation (C_D contains an instance of C), then no change to C_D's definition is implied. We discuss this, as well as other details such as method table updates, further in Section 3.3.

The two conditions for $\mapsto$ preserve type safety. The first condition prevents static type violations, and the second prevents dynamic type violations. No other constraints are needed. After any number of changes, a program is still type-safe. Formally, we state this as a theorem:

Theorem 1. *Given $\underline{P} \overset{*}{\mapsto} \overline{P}$, if $\underline{P}$ is type-safe, then $\overline{P}$ is type-safe.*

We prove Theorem 1 using induction on the number of class changes enacted.

Base step: if no change has been made to P, then P is type-safe. True by the definition of a valid Java program.

Inductive step: If $\underline{P}$ is type-safe, then $\overline{P}$ is type-safe. We prove this using contradiction: we have some $\underline{C} \mapsto \overline{C} \implies \underline{P} \mapsto \overline{P}$, where $\underline{P}$ is type-safe and $\overline{P}$ is not. Therefore, $\exists$ some class $X \in P$: $\underline{C} \mapsto \overline{C} \implies \underline{X} \mapsto \overline{X} \wedge \overline{X}$ is not type-safe. There are two cases:

Case 1: $\overline{X}$ contains a static type violation: $\exists\, Y$: $\underline{C} \mapsto \overline{C} \implies \underline{Y} \mapsto \overline{Y} \wedge X \propto (\underline{Y} \ominus \overline{Y})$. Recall Condition 1, which requires that $\forall X \in P : \neg(\exists\, Y, X \propto (\underline{Y} \ominus \overline{Y}))$. This condition contradicts the above.

Case 2: $\overline{X}$ contains a dynamic type violation: $\exists\, C_D : C_D \neq C : C_D \propto T$. $\tau(\underline{X}) \not\subseteq \tau(\overline{X})$. However, $\underline{X} \mapsto \overline{X} \iff \neg(\exists\, C_D : C_D \neq C : C_D \propto T)$ by Condition 2 of $\mapsto$, and we have a contradiction.

Therefore, if $\underline{P}$ is type-safe, then $\overline{P}$ is type-safe. □

2.2 Support for Dynamic Classes

Dynamic classes can be implemented in several ways: (i) by changing the Java language to support mutable classes, as done in [8], (ii) using library-based support, as done with C++ in [19], or (iii) by modifying the virtual machine. We did not wish to modify the syntax or semantics of the Java language. The library-based solution is inefficient and contains intractable implementation problems. In this section, we describe our design, which uses a modifed virtual machine to provide runtime system support for dynamic classes, and extends the class loader to provide an interface.

Java class loader: The interface by which users manipulate dynamic classes is an extended Java class loader. Thus, we begin our discussion with some pertinent background on the Java class loading mechanism.

```
public abstract class ClassLoader {
  public Class loadClass(String name);
  protected Class findClass(String name);
  protected Class defineClass(String name, byte[] b, int off, int len);
  protected void resolveClass(Class c);
        ...
}
```

Fig. 1. Java VM class loader

The JVM resolves references to a class during runtime using a mechanism called the *class loader* [28]. A class loader is responsible for locating the definition of a class, which takes the form of a class file, and loading it into the JVM. A class in Java is, thus, defined by both its name *and* the class loader that loaded it. The JVM defines two kinds of class loaders: the system class loader and user-defined class loaders. The system class loader is the default class loader used for locating and loading system classes and user-defined classes. Users can override the behavior of the default class loader by defining their own class loaders. To build a specialized class loader, the user must extend the abstract base class `ClassLoader`. Figure 1 depicts part of the interface of `ClassLoader`, as well as the methods that can be overriden in user-defined subclasses.

The dynamic class loader: The programming interface for dynamic classes is the *dynamic class loader*. This class, `DynamicClassLoader`, extends the JVM class loader. In addition, it supports replacement of a class definition, and update of objects and dependent classes. Any class loaded by an instance of `DynamicClassLoader` is automatically a dynamic class.

We chose this approach for several reasons. Since the class loader loads, stores, and examines class definitions, it is a logical choice for a module that modifies class definitions. The design extends Java's dynamic linking mechanism, instead of replacing it. Thus, it supports existing code, with little or no modification. Users can choose to use dynamic classes when and where they see fit. Most importantly, our design preserves the security mechanisms inherent to the class loader system, which include namespace separation and bytecode verification.

```
public class DynamicClassLoader extends ClassLoader {
  public Class reloadClass(String newc);
  public final int replaceClass(String oldc, Class newc);
    ...
  // several overloaded versions of replaceClass are defined
  // for convenience
}
```

Fig. 2. `DynamicClassLoader` interface

The dynamic class loader loads classes from disk in the same manner as the system class loader. It complies fully with the specified semantics of a Java class loader, as described above. However, the dynamic class loader provides additional methods (`reloadClass` and `replaceClass`) that can reload an active class and replace it with a new version. Using runtime system support, these methods implement the semantics of class change ($\mapsto$) as stated in Definition 15. Method `reloadClass` is similar to `loadClass` in that it reads a designated class file from the disk, creates a class object, and returns it. However, `loadClass` does not load classes that are already defined in the system, whereas `reloadClass` succeeds whether the target class was previously defined or not. Given $\overline{C}$, `replaceClass` defines $\overline{C}$ to be the new definition of C, and initiates instance update. These rely on several native methods that interface with the VM's internal data structures.

We provide relevant implementation details in Section 3. Figure 2 summarizes the interface to `DynamicClassLoader`.

Users can extend the dynamic class loader by redefining `reloadClass` or `findClass`. Method `replaceClass` is a `final` method and cannot be overridden. This ensures consistent class redefinition and security, as `replaceClass` performs verification of $\overline{C}$, and enforces namespace constraints.

2.3 Security

In Java 1.2, the JVM prevents classes from performing forbidden actions by using bytecode verification, supporting namespace partitioning, and enforcing user-defined access control policies. The bytecode verifier examines each class before loading it into the JVM, checking for type violations and other illegal operations. Figure 3 depicts a typical namespace configuration in a system that hosts mobile, untrusted applets, such as a web browser. Applets are each run in their own namespace, defined by separate class loaders. Resource classes provided by the host are placed in another namespace Access between namespaces is only permitted `down` the tree; applets are effectively isolated from one another. Furthermore, the user can specify access control policies for more fine-grained protection. In Sun's JDK 1.2 security model, the class `AccessController` acts as a security monitor [2,14,15]. All protected resources must call `AccessController.check()`, which checks the access against the permissions specified in the security policy. Although the security policy, and thus permissions, can change, the set of protected resources is static.

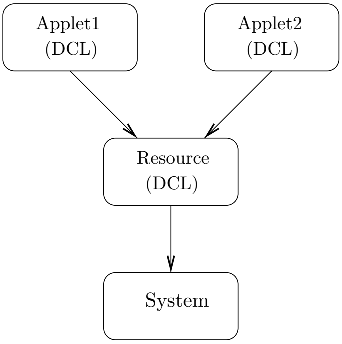


Fig. 3. Typical namespace configuration. `DCL` indicates a dynamic class loader.

Dynamic classes pose new security hazards. Malicious code could potentially bypass many existing security mechanisms, by modifying either itself or the protected classes it targets. Specifically, a malicious class could modify itself in order to perform forbidden actions, or modify sensitive classes to either perform or allow forbidden actions. Consider, for instance, Figure 4. A host provides a resource, `myResource`, to which access is restricted via an access control policy. A malicious applet, evilApplet, contains code that replaces the protected resource with a new version that does not invoke the access controller. evilApplet can then gain access to which it is not entitled.

```
public class myResource { // original version
  public static void foo() { // protected resource method
    // perform security check
    accessController.check(new myResourcePermission());
    // access sensitive resource
    ...
  }
}
public class myResource { // weakened version
  public static void foo() { // method now unprotected
    // skip security check...
    // access sensitive resource
    ...
  }
}
public class com.evilDomain.evilApplet extends Applet {
  // malicious mobile applet
  public void start() {
    // get handle to dynamic class loader
    DynamicClassLoader dcl = getClass().getClassLoader();
    // get a URL class loader, linked to originating, evil host
    URLClassLoader ucl = new URLClassLoader(‘‘evil.domain.com’’);
    // use it to load a weakened version of the resource class
    Class wr = ucl.loadClass(‘‘myResource’’);
    // now use dcl to replace protected resource with weakened version
    dcl.replaceClass(‘‘myResource’’, wr);
    // invoke resource, which should be denied but won’t be
    myResource.foo();
  }
}
System security policy: only allow local namespace access to resource
grant codebase ‘‘localhost’’ {
  permission myResourcePermission;
}
```

Fig. 4. Using dynamic classes to bypass access control.

We want to ensure that dynamic classes do not introduce any new security risks. Therefore, Java’s security mechanisms must extend to dynamic classes. This requires several measures.

The dynamic class loader subjects all modified classes to bytecode verification before loading them into the JVM, so a malicious class cannot instrument

```
grant codebase ‘‘localhost’’ {
    permission ucd.pdclab.dynclass.modifyClassPermission;
}
```

Fig. 5. Grant class modification privileges *only* to classes in the local codebase.

itself to include illegal bytecode operations. The dynamic class loader honors the separation between namespaces by replacing only those classes defined within its own namespace. Returning to Figure 3, let DCL denote a dynamic class loader. Thus, applets and resources are dynamic classes. An applet running in namespace Applet1 cannot use its own class loader to replace a class defined in Applet2. The scenario depicted in Figure 4 cannot happen.

These steps do not, however, prevent a malicious applet in Applet1 from invoking the *resource* namespace dynamic class loader and modifying resource classes. Thus, dynamic class loaders should be protected by an access control policy. Figure 5 contains a simple example of such a policy: only classes from the local codebase, or namespace, can invoke the dynamic class loader. Applets are excluded. DynamicClassLoader contains appropriate calls to the access controller, as described in earlier. Under this policy, applets in Figure 3 cannot modify system or resource classes, nor can they modify themselves. A similar policy could provide full protection for myResource in Figure 4.

In practice, it is possible to violate Java's type model and compromise security [37]. This is due to problems in the semantics of dynamic linking and the implementation of the virtual machine. The issue does not bear directly upon dynamic classes, and we do not address it.

Another compelling question is that of the security behavior of the program itself. Ideally, we could like to ensure that the security behavior of $\overline{C}$ is no weaker than that of $\underline{C}$; that is, no potential security holes are introduced into the code. However, this problem is unsolvable in the most general case. Conceivably, heuristics could be used, together with assumptions about or constraints on program behavior to solve the problem for specific cases. Such heuristics are, however, beyond the scope of this paper. It remains the responsibility of the programmer to maintain security behavior across changes.

3 Implementation

DynamicClassLoader requires virtual machine support for reloading a class definition, finding and updating dependent classes, and finding and updating instances of modified classes. We have modified the Solaris version of Sun's JVM (JDK 1.2). Much of our discussion here pertains specifically to that VM. Our implementation includes a shared library containing functions that support class replacement and instance update. We have also made minor changes in some data structures and functions internal to the JVM to support the library functions. In the remainder of this paper, we refer to this modified, dynamic classes-enabled virtual machine as DVM.

Adding support for dynamic classes requires understanding and manipulating the JVM's internal data structures and functions in several areas. The JVM uses several optimizations to increase performance, and we take this into account in our design. In this section, we first provide the necessary background on the JVM. We then discuss our implementation.

3.1 Background: Java Virtual Machine

Here we describe the general architecture of the JVM. We focus only on those aspects of the architecture that are relevant to to support for dynamic classes. Specifically, we describe JVM's runtime memory organization, the structure and function of class definition objects, and the optimizations within the bytecode interpreter.

Execution of Java programs: Java programs are composed of classes, each of which is stored in a separate class file. A class file contains the types and definitions of fields and methods defined in the class. All references to classes, fields, or methods are symbolic and contain enough information to allow the JVM to link classes in a type safe manner.

Java heap organization: All Java objects are allocated within a data structure known as the Java heap. In many JVM implementations, including JDK 1.2, the heap is divided into a handle pool and an object pool. Java objects are always addressed indirectly through their handles. The use of handles facilitates garbage collection. When an object is moved, only the pointer in its corresponding handle needs to be updated; the handles never move.

This model is very useful when handling object update for dynamic classes, as described in Section 3.2. The DVM can allocate new space for an object when updating it, without changing the handle used to reference the object.

Class objects: A class object, an instance of `Class`, is created for each loaded class. This object contains the entire class definition, including field types, method signatures and bytecode, and inheritance information. Class objects are special in that they are allocated on the Java heap, but some fields contain pointers into the interpreter's C++ heap. Thus, the code segment for an executing program is distributed among several Java class objects. All names – or classes, methods, fields, etc. – used by the class are stored in the constant pool. In the bytecode, indices into this constant pool are used as symbolic references. Our implementation uses the semantic information contained in class objects to assess dependency relationships among classes and methods.

JVM optimizations: The JVM performs several optimizations that can obfuscate internal data structures and cause problems during class changes. These optimizations include the use of method tables, inlining, quick instructions, and direct referencing. Below, we describe the problems that the optimizations raise during dynamic class implementation and how we resolve them.

Each class data structure contains method and field tables used by virtual method calls and other instructions. These tables contain the names of all methods or fields defined within a class C and its superclasses; each entry has a pointer to the method body or field visible in C's scope. When changing C, the DVM rebuilds the method tables in C and all of its subclasses.

When a class is first loaded, its constant pool contains symbolic references, and its bytecode contains indices into the constant pool for all method and data access instructions. The first time the JVM encounters any such instruction, it checks if the constant pool entry has been resolved, and resolves the entry if needed. Then, the JVM changes the instruction to a special *quick* instruction that does not perform the check. Any subsequent execution of that instruction is relatively fast. Certain quick instructions contain offsets into objects or method tables that may change when a class is modified. To make class updating cleaner, the DVM only uses quick instructions that do not contain any offsets or direct references. This avoids the need to update bytecode, but incurs a slight performance penalty.

JDK 1.2 includes a Just-in-Time (JIT) compiler [22]. JIT compilers provide a significant speed boost to a Java VM by generating native machine code from Java bytecode on the fly. This optimization has an impact on dynamic classes – if a method is modified, previously generated machine code becomes invalid. Therefore, if the JIT compiler is enabled, the DVM must ensure that any modified methods are recompiled. We have not yet implemented this step. At present, the JIT compiler is disabled within the DVM.

The JVM also performs inlining, where some method invocation instructions are replaced by the actual bytecode of the method called. This technique also affects dynamic classes, as inlined code may be invalidated by a class change. We have, therefore, disabled method inlining for all classes loaded by a dynamic class loader. System classes and non-dynamic classes are inlined as usual. We plan to re-enable inlining for dynamic classes by forcing a recompile of any methods that contain inlined code for methods that have changed.

3.2 Updating Instances

There are several alternatives for handling existing instances when a class changes: none, some, or all of them can change to match the new definition. We discuss the options, and justify our decision to enforce global update. Then, we address the implementation details involved in finding, locking, and updating the objects.

Instance update models: Possible models for instance update include a version barrier, passive partitioning, global update, and active partitioning. We describe and compare these models here. Our definitions of version barrier, passive partitioning, and global update, as well as Figure 6, are based on [19].

First, the DVM could use a barrier on object versions. With this solution, $\underline{C} \mapsto \overline{C}$ cannot occur until all objects defined by $\underline{C}$ have expired, as shown in Figure 6(a). Note that, in this case, t_{update} is delayed until all old objects have expired. This solution lacks the flexibility we desired. Effectively, active classes cannot change.

Another possibility is passive partitioning, where objects created before $\underline{C} \mapsto \overline{C}$ are unchanged, and any created afterwards reflect the new type. Figure 6(c)

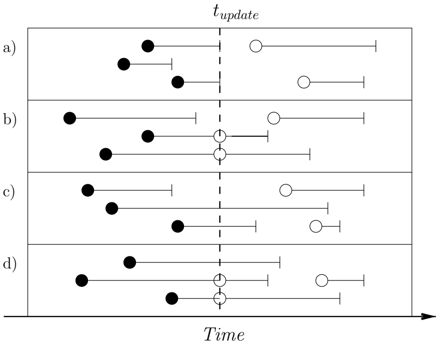


Fig. 6. Object update models: (a) version barrier, (b) global update, (c) passive partitioning, and (d) active partitioning. Black objects denote old version, and white are new version.

depicts this model. In this case, as with the previous, multiple definitions of a class can be active simultaneously. This breaks the Java name-binding semantics, and introduces ambiguity that we wished to avoid.

Active partitioning allows the user to actively select which objects to update and which to leave at the previous version, thus partitioning the objects into type spaces. Such a model effectively implements fully dynamic typing, as one could redefine classes at the granularity of individual objects, giving each object its own dynamic type descriptor. Figure 6(d) illustrates this model. As discussed in Section 2, we chose not to make such a drastic change in Java's type system.

Therefore, our model uses the fourth method: global update of all objects defined by C, shown in Figure 6(b). We defined $\mapsto$ (see Definition 15) such that the DVM must locate and update all instances of C and its subclasses to reflect the new definition, $\overline{C}$.

Implementing incremental global update: Once the DVM has determined that a modified class's instances must be updated, the problem remains of actually locating and processing them. This problem is similar to that of garbage collection. In both cases, there are three major steps: find relevant objects on the heap, lock them, and process them.

Garbage collection algorithms generally fall into one of three categories: basic, incremental, and generational [43]. Basic algorithms use techniques such as reference counting or mark and sweep to identify and process objects in a single transaction. This transaction is atomic in that all other threads must block while garbage is collected, and has the undesirable effect of temporarily halting program execution. Incremental algorithms interleave garbage collection with program execution, alleviating the pause effect. Generational algorithms exploit temporal locality in memory usage to optimize garbage collection. We required a more efficient method than a basic algorithm, and generational algorithms rely on assumptions about program behavior that may not apply to instance update. Therefore, we chose an incremental mark-and-sweep approach to updating. In this method, there are two phases: the *mark* phase, during which objects are identified, and the *sweep* phase, in which they are actually updated. The mark phase is atomic, and the sweep phase proceeds incrementally.

In the mark phase, the DVM finds the objects by scanning the handle pool, looking for instances of C. When it finds one, the DVM sets a bit in the object header to indicate that the object needs to be updated. Finally, the DVM modifies the class object pointer present in the corresponding handle structure to point to the new definition, $\overline{C}$. The mark phase eliminates the need to check all object references, thus increasing overall efficiency – the DVM only traps references when updates are pending.

In the sweep phase, the DVM incrementally updates marked objects. To maintain the heap in a consistent state, the DVM traps all accesses of marked objects. If any objects need to be updated, the DVM checks the update bit of the target object when interpreting an object reference instruction. It may seem that this sweep technique implements version partitioning in that old and new versions may actually be present on the heap at the same time. However, the implementation guarantees that any old object will be updated *before* it is referenced. The state of an inactive object does not matter. An advantage of this technique is that the DVM does not update objects destined for garbage collection. The disadvantage is a slight performance cost.

The DVM takes several steps to update an object O. Since other threads may be active, it first locks O to prevent race conditions. Then, processing may continue. The DVM allocates a new object $\overline{O}$, where $\overline{C} \geq_O \overline{O}$, and copies $\underline{O}$'s data to $\overline{O}$. It initializes any *new* fields within $\overline{O}$ to zero (`null`), then switches the handles of $\underline{O}$ and $\overline{O}$. Any references to $\underline{O}$ now point to $\overline{O}$, and O is reclaimed by the garbage collector. Finally, the DVM unlocks O, and the method that triggered the update may continue.

3.3 Updating Dependent Classes

When redefining C, the DVM changes the state of all dependent classes to enable C's new definition. This process requires several steps. First, the DVM identifies all dependents and categorizes them by their relation to C – subclass, method usage, etc. It re-resolves all dependent classes, and updates additional information within subclasses.

The DVM identifies dependent classes by scanning the constant pools of all loaded classes for C. Then, it updates each one according to its relation to C. When a class is first loaded, its constant pool contains symbolic references to other class objects and their methods and fields. When the JVM resolves the class, it replaces these references with actual pointers to the referenced object. When the DVM redefines C, any pointers to $\underline{C}$ become invalid, and the DVM replaces all resolved references with the original symbolic references. It then resolves the class and replaces the references with pointers into $\overline{C}$. To support restoration of symbolic references, we added an additional field to the class object structure that contains the original constant pool.

C's subclasses require additional processing. If any data or methods are added to C, the DVM updates the method and field tables of all subclasses. It rebuilds these tables when it re-resolves a class. Classes that contain instances of C as data require no further action. Since Java objects contain references in their component fields and not entire objects, classes are not affected by a change in the class definition of one of their components.

3.4 Pitfalls in Dynamic Classes

The introduction of arbitrary new code into a running Java application has many potentially negative consequences. Type safety may be affected, race conditions may result, etc. In this section, we analyze several problems that we encountered during the implementation and the present the solutions.

Type safety: Recall from Section 2 that the operation $\underline{C} \mapsto \overline{C}$ is allowed if it does not cause static or dynamic type violations. Therefore, before making any change, the DVM verifies these conditions. This maintains program correctness and type-dependent security mechanisms.

The DVM checks for static type violations by examining $\underline{C}$, $\overline{C}$, and any dependents. Assume that $\underline{C}$ has a field X, and the switch to $\overline{C}$ removes X. There are two possible cases for invalid references – within C, and in other classes. Since we enforce the constraint that $\overline{C}$ must be a valid class definition, the first case is impossible. Before enacting the change, the DVM resolves $\overline{C}$ and runs through the bytecode verifier. Therefore, whether $\overline{C}$ is a compiled class or was generated on the fly, it cannot contain any references to X. However, we may have another class C_D that is dependent on C and references X. In Section 3.3, we described how to identify dependent classes quickly by scanning their constant pools. We extend this technique to locate references to deleted fields or methods such as X – the name $C.X$ must be present in C_D's constant pool. If any such references are found, the DVM invokes a user-supplied handler, passing it a list of classes that depend on $\underline{C}$. This handler may then throw an exception, update dependent classes if possible, etc. This step ensures that all classes defined, and thus the program itself, are valid after the change.

Likewise, the DVM checks the second condition by comparing $\underline{C}$ and $\overline{C}$, and recursively examining C's superclasses. If $\underline{C} \overset{*}{\sqsubseteq} C_S$ and $\overline{C}$ does not, the DVM searches for any classes that depend on C_S. If any are found, the DVM throws an exception. Similar steps are taken if $\underline{C} \overset{*}{\geq}_I T$ and $\overline{C}$ does not. It is a straightforward matter to extract this information from the class object data structures.

Race conditions in multithreaded applications: Multithreaded applications raise the issue of race conditions on the definitions and instances of dynamic classes. During redefinition, the data in $\underline{C}$ and $\overline{C}$ are in an inconsistent, transitory state. If an active thread references C during this time, runtime system errors will likely result. The DVM prevents this event by blocking all threads prior to performing the replacement.

Ideally, the DVM could identify all threads that depend on C, and block only those threads. Unfortunately, this requires a lengthy recursive search of all loaded classes for every frame on every thread stack. This operation is actually much more expensive than the class replacement, which is fairly brief. Thus, the DVM blocks all threads except for the thread performing the change, and allows the threads to continue after the change is complete.

Native methods: The JVM allows users to run native methods, and the potential for race conditions during a class change exists here as well. Since native methods do not use a Java stack, and their code cannot be easily examined for dependencies, it is very difficult to determine if it is safe to make a change while a native method is active. Further, native code does not consist of discrete bytecode instruction sections. It is difficult to determine when it is safe to block a native method without causing race conditions as described above.

One solution is to simply disallow class changes while native methods are active. Unfortunately, many native methods are involved in I/O and include polling loops; they are perpetually active. Therefore, the DVM does not block native threads, nor does it wait for them to finish or reach any particular state before continuing with a class change. There is a danger that a native thread could access some internal data structure while the DVM is modifying a class. However, since the JVM cannot control the execution of native methods, there is always the danger that one will corrupt the runtime state in some manner. We assume that all native methods are trusted to behave properly.

Race conditions during object update are easier to handle. Native methods should "pin" Java objects before accessing them, a form of locking. Before changing a class, the DVM scans the heap and ensures that no instances of that class are pinned.

Changing active methods: An interesting problem involves changing a method *that is currently running.* Given a method $C.M$, we must first determine if $C.M$ has changed. Whenever the dynamic class loader loads a class, it calculates and stores a hash value for each method. The DVM can then determine if M has changed by comparing the old and new hash values.

This cannot be done by a simple string compare of $\underline{C}$'s and $\overline{C}$'s versions of M, since the constant pools indices used as arguments in the bytecode may change, even if the method code does not. Any deeper examination of the bytecode becomes costly. So, whenever the dynamic class loader loads a class, it calculates and stores a hash value for each method. This hash value includes all bytecode instructions, and the full names of all classes, methods, and fields referenced, rather than the symbolic references. Then, the DVM can determine if M has changed by comparing the old and new hash values. There is a slight possibility of collision, where two different methods give the same hash value, thus causing a false negative. Therefore, in the event of a match, the DVM also checks other information such as bytecode length and stack size.

Once it has determined that M has changed, the DVM must include $C.M$ in its search of the active thread stacks. Given that M is at an arbitrary point in execution, that Java bytecode contains no semantic information about control flow, and that no particular relationship between $\underline{M}$ and $\overline{M}$ is required, it is impossible, in the general case, to determine where and how to continue execution in $\overline{M}$. For instance, if $\underline{M}$ and $\overline{M}$ solve the same problem using different algorithms, there may not be a point in $\overline{M}$ corresponding to the current location in $\underline{M}$. Or, $\overline{M}$ may use local data that is not present in $\underline{M}$, and that must be initialized. This problem is similar to that posed by security behavior across class changes, as discussed in Section 2.3. Again, heuristics might be used to solve specific cases, but such heuristics are beyond the scope of this paper. Therefore, active methods cannot be changed. If the user attempts to change an active method, the DVM throws an exception, aborting the offending thread. The user may handle this exception in another manner, by continuing the thread but aborting the replacement, terminating and re-invoking the method, etc.

4 Discussion

In this section, we first analyze the performance of the DVM, as compared to the standard JVM. We then compare our design and implementation with other work related to dynamic evolution.

4.1 Performance Analysis

We are concerned with two performance factors: baseline performance of the modified VM, and the cost of replacing a class and updating its instances. We have performed a series of experiments to determine precisely where penalties are incurred and their degree, and to suggest optimizations and improvements. These results pertain to an unoptimized DVM; work on optimization is proceeding apace.

Overhead of adding dynamic classes to JVM: It is straightforward to test the baseline performance of the DVM, simply by running a series of benchmark programs on both the DVM and unmodified JVM. We ran the SpecJVM '98 benchmark suite [41], with a problem size of 100, on a 266 MHz Intel Pentium II running SunOS 5.6. Figure 7 summarizes the results. The performance penalty varied between applications from around five percent to nearly ten, with the average around six percent.

SpecJVM Programs	*JVM*	*DVM*	*JVM/DVM*	*DVM w/ repl.*	*no repl./repl.*
jess	1420.888	1562.581	90.9%	1738.559	90%
db	2675.772	2932.931	91.2%	3257.733	90%
javac	1692.285	1840.9	91.9%	2181.056	84%
mpegaudio	6383.705	6743.099	94.7%	6853.353	98%
mtrt	1709.399	1883.119	90.8%	2163.25	87%
jack	2083.441	2306.306	90.3%	2559.645	90%
Total	15966.552	17269.977	92.5%	18753.596	92%

Fig. 7. SpecJVM benchmark results. All time in seconds.

We ran another experiment to determine the penalty caused by each of our modifications. For this experiment, we used a simpler set of benchmark programs [16], run with different versions of the DVM. Each successive DVM version

VM version	*time*	*JVM/DVM*	*penalty*
JVM	54.6	–	–
DVM	55.8	97.8%	2.2%
No quick instructions	58.8	92.9%	4.9%
Update object check	58.9	92.7%	0.2%
Class replace lock check	62.4	87.5%	5.2%

Fig. 8. Performance cost distribution.

activates an additional instrumentation of the unmodified JVM. Instrumentations include the elimination of quick instructions, checking if an update is needed in object reference instructions (see Section 3.2), and the class replace lock check for object reference and method invocation instructions. Figure 8 summarizes the performance cost distribution. The costly modifications are the elimination of quick instructions and the class replace lock; each incurs an approximately 5% penalty. The penalty caused by the object update check is very small. Current efforts focus on reducing these penalties by implementing a more efficient locking mechanism, and possibly re-enabling quick instructions for non-dynamic classes.

These data inform the wide range in performance cost reported in Figure 7. Applications that have a higher proportion of object reference and method invocation bytecode instructions, as compared to other instruction types, suffer more from both the loss of quick instructions and the class replace lock check.

Cost of modifying classes: The acquisition of meaningful data about the cost of replacing a class and updating instances is more complex. Many variables are involved, including the behavior of the application (object allocation and usage, etc.) and the state of the runtime system (number of classes loaded, thread state, etc.). Thus, different applications will generate widely varying data. We have experimentally modeled this cost by running the Spec benchmarks, as above, alongside a thread that periodically replaced a randomly selected user class. We did not modify any Spec classes; any such class is replaced with itself, causing no instance update. We included a "dummy" class that, when changed, has a different implementation. Our extra thread allocates and periodically accesses many instances of this class. The number of objects used in this set of experiments was 10000, and the interval between class changes was 5 seconds – we consider this to be a fairly heavy replace/update load. We show the results in Figure 7. The overall performance penalty ranged from ten to sixteen percent, with average at eight percent.

4.2 Related Work

We survey related work in dynamic evolution in the context of programming models. We loosely classify techniques according to the semantics of changing code and the programming interface.

Dynamic classes: Under dynamic classes, the definition of a type may be changed at runtime. However, the defining type of an individual object may not, as is the case with dynamic typing. Any change is applied directly to the type definition rather than its instances. Therefore, objects in memory must somehow be partitioned between different versions of the class.

C++-style templates, at first glance, seem to provide some dynamic capability – a template class or function can change based on what template parameter is provided. However, this is static. Effectively, templates generate new classes during compilation, but cannot generate or modify classes at runtime. The Java interface construct suffers from similar limitations, as discussed under dynamic linking. The Java interface construct is not sufficient either; one may load and use a new implementation class for an existing interface, but any existing instances of the original implementation are not affected.

Hjalmtysson and Gray [19] implement dynamic classes in C++. The system uses a wrapper, or proxy class, method that essentially implements Java style interfaces in C++, and further extends the mechanism to allow linking of a new implementation class at runtime, and the presence of multiple active versions. This does not require runtime system support or language extensions, and could be applied to Java as well — we chose not to do so for performance reasons.

Shadows [12] is a system for projecting objects between type spaces, and has been implemented in C++. Shadows also uses a form of proxy class, called a *shadow map*. This map is used to map nodes from the original data structure or type into an extended structure, or *shadow*. Shadows uses runtime type checking to maintain type safety. As with dynamic C++ classes, Shadows does not require compiler or runtime support, but can only be used with specifically coded programs and incurs overhead that might be prohibitive in a Java environment.

Delegation [30] provides a mechanism by which Kniesel [26] implements dynamic classes. Delegation permits *object*- rather than class-based inheritance. A class can contains *delegates*, which are objects invoked to perform certain functions. By changing the delegates bound to a function, one can easily change that function's implementation.

Dynamic linking: Dynamic linking [20,24,11] allows names to be bound when the program begins execution. Once done, this binding cannot be changed without restarting the program. The common point among all traditional stages of binding is that any type or method name can only be bound *once* across all phases. Further, dynamic linking contains no notion of state or correctness. Even if it were possible to re-link a dynamic library, there is no semantic framework dictating how and when it may be done.

Load-time transformation: Several projects exist that support modification or generation of classes at load-time (before or during class loading). This technique can be used to optimize or reconfigure applications by generating and loading specialized classes. However, the method is subject to the limitations of dynamic linking. New classes can be generated and loaded, but classes and objects previously present in the JVM are not affected. Classes can change in the static or loaded state, but not while active. Linguistic reflection [25], Binary Component Adaptation [23] and JOIE [7] implement load-time transformation.

Dynamic architectural frameworks: Architectural frameworks such as COM [5], CORBA [1] and C2 [42] provide a mechanism by which a program can be described in terms of high-level components such as modules and connectors. In general these frameworks are static – once defined, a program is static and its design cannot be changed at runtime. Dynamic frameworks allow the user to change the high-level architectural specification of a program at runtime.

Archstudio [34] provides graphical and command-line tools used to modify a C2-Java program specification at runtime. An attempt to change the specification invokes an Architecture Evolution Manager, which checks the request for validity, and modifies the program's implementation accordingly.

The Argus language [31], which provides a client/server model for distributed computing, supports dynamic update of servers, or guardians [4]. Similarly, Conic [32] provides a module-based environment using message passing. Modules communicate via ports, and may be dynamically updated by switching all links from the present version of a module to a new one. However, the ports between modules are static, thus connections cannot be created or broken dynamically.

Dynamic typing: CLOS [40] and Smalltalk [13] support dynamic typing, in which the type descriptor of an object may be changed freely at runtime. Method code may be modified, data fields and methods may be added or removed, etc. For example, the Information Bus [33] distributed systems architecture uses a CLOS-derived language to implement dynamic classes. Fabry [10] implements a dynamic type system using capabilities. Widening [39] provides a mechanism for *constrained* dynamic type changes, in which objects may be temporarily "widened" to a subtype of their defining class. [8] implement a mechanism similar to widening, for imperative languages, and present a formal type system with proof of soundness.

Dynamic typing, in its unconstrained form, supports the greatest flexibility. However, static type checking of any kind becomes infeasible, so the runtime system must support complete runtime type checking, with all associated overhead.

Parallel versions: One approach to replacing one version of a program ($\underline{P}$) with a new version ($\overline{P}$) is to beginning running *both* versions in parallel, transferring $\underline{P}$'s state to $\overline{P}$ at an appropriate time. Both software and hardware-based solutions exist. Gupta and Jalote [17] use processes as update vectors, and SCP [38] uses redundant CPUs.

While efficient, redundant hardware is obviously expensive, and only practical in certain situations such as the telecommunications environment towards which SCP is targeted. Parallel processes are an efficient technique. However, transfer of state, which may include open files, displays, and elements not affected directly by the change, can be awkward.

5 Conclusion

We have described the design and implementation of dynamic classes in Java, using runtime support. Our solution is novel in the combination of type safety preservation, nearly unrestricted changes, support for any Java class, and efficiency. These features balance efficiency, convenience, safety, and power of expression.

We have developed a dynamic security infrastructure using dynamic classes [18], as well a mechanism that enhances the dynamism of JDK 1.2's native security model. We are also working on a dynamic architectural framework based on Java Beans [21], and a code distribution mechanism. These applications, in conjunction with our performance analysis, show that dynamic Java classes are a useful language extension that supports an exciting class of software systems. Further optimization of the DVM is an ongoing process.

Currently, our primary focus for future work is the extension of the dynamic classes model to distributed systems. The introduction of distributed applications running across multiple hosts, with objects migrating between them, has many implications. For example, due to latency and packet dropping over the network, our current synchronization model does not scale well to multiple hosts. It is difficult to avoid race conditions while maintaining efficiency. One solution is to simply accept race conditions and work around them. This approach implicitly creates a multiple-version model of classes, which merits further examination.

Acknowledgements. We would like to express our appreciation toward Brant Hashii, David Peterson, and Michael Haungs for their support and assistance. We also thank the anonymous reviewers for their excellent comments and suggestions.

This work is supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Project Agency (DARPA), Rome Laboratory, or the U.S. Government.

References

1. The Common Object Request Broker: Architecture and Specification, Revision 2.0. Object Management Group, July 1996. http://www.omg.org/corba/corbiiop.htm.
2. James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol. II, Electronic Systems Division, Air Force Systems Command, Hanscom AFB, Bedford, MA 01731, October 1972. [NTIS AD-758 206].
3. K. Arnold and J. Gosling. *The Java Programming Language.* Addison Wesley, 1996.

4. T. Bloom. *Dynamic Module Replacement in a Distributed Programming System.* PhD thesis, MIT, 1983.
5. K. Brockschmidt. *Inside OLE 2.* Microsoft Press, 1994.
6. Eduardo Casais. Managing class evolution in object-oriented systems. In *Object-Oriented Software Composition.* Prentice Hall, 1991.
7. Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX Annual Technical Symposium*, 1998.
8. Sophia Drossopoulou, Mariangiola Dezani-Ciancaglini, Ferruccio Damiani, and Paola Giannini. Objects dynamically changing class. August 1999.
9. Sophia Drossopoulou, Tanya Valkevych, and Susan Eisenbach. Java type soundness revisited. October 1999.
10. R. S. Fabry. How to design a system in which modules can be changed on the fly. In *2nd International Conference on Software Engineering*, 1976.
11. Michael Franz. Dynamic linking of software components. *IEEE Computer*, 18(9162):74–81, March 1997.
12. Jonathan J. Gibbons and Michael J. Day. Shadows: A type-safe framework for dynamically extensible objects. TR TR-94-31, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, 1994. Available from www.sunlabs.com.
13. Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison Wesley, Menlo Park, CA, 1983.
14. L. Gong. Java security: Present and near furture. *IEEE Micro*, 17(3):14–19, May-June 1997.
15. L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, December 1997.
16. William Griswold and Paul Phillips. Bill and Paul's Excellent UCSD Benchmarks for Java (version 1.1). UCSD Software Evolution Group. http://www-cse.ucsd.edu/users/wgg/JavaProf/javaprof.html.
17. Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software – Practice and Experience*, 23(9), September 1993.
18. B. Hashii, S. Malabarba, R. Pandey, and M. Bishop. Supporting reconfigurable security policies for mobile Java programs. In *Proceedings of WWW9*, May 2000. To appear. Currently available at http://pdclab.cs.ucdavis.edu.
19. Gisli Hjalmtysson and Robert Gray. Dynamic C++ classes: A lightweight mechanism to update code in a running program. In *Proceedings of the USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998. USENIX.
20. W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *SOFTWARE–Practice and Experience*, 21(4):375–390, April 1991.
21. JavaSoft. Component-based software with JavaBeans and ActiveX. White paper.
22. JavaSoft. *The Java Native Code API.*
23. R. Keller and R. Hölzle. Binary component adaptation. In *ECOOP'98 Proceedings*, Lecture Notes in Computer Science. Springer Verlag, 1998. Also available at http://www.cs.ucsb.edu/oocsb/papers/TRCS97-20.html.
24. James Kempf and Peter B. Kessler. Cross-address space dynamic linking. TR TR-92-2, Sun Microsystems, 2550 Garcia Avenue, Mountain View, CA 94043, 1992. Available from www.sunlabs.com.
25. Graham Kirby, Ron Morrison, and David Stemple. Linguistic reflection in Java. *Software-Practice and Experience*, 28(10), 1998.

26. Gunter Kniesel. Type-safe delegation for run-time component adaptation. In *European Conference on Object-Oriented Programming.* Springer, 1999.
27. Robert Laddaga and James Veitch. Dynamic object technology. *Communications of the ACM*, 40(5):36–38, March 1997.
28. S. Liang and G. Brach. Dynamic class loading in the java virtual machine. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, number 10, Vancouver, October 1998. ACM.
29. S. Liang and G. Bracha. Dynamic class loading in the Java Virtual Machine. Draft. JavaSoft, Sun Microsystems, April 1998.
30. Henry Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. In *OOPSLA*, 1986.
31. B. Liskov. Distributed programming in Argus. *Communications of the ACM*, March 1988.
32. J. Magee, J. Kramer, and M. Sloman. Constructing distrubuted systems in Conic. *IEEE Transactions on Software Engineering*, June 1989.
33. Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus – an architecture for extensible distributed systems. *ACM Operating Systems Review*, 27(5):58–68, December 1993.
34. Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering*, 1998.
35. R. Pandey and B. Hashii. Providing fine-grained access control for Java programs. In *13th Conference on Object-Oriented Programming. ECOOP'99*, Lecture Notes in Computer Science. Springer-Verlag, June 1999.
36. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.
37. Vijay Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. http://www.research.att.com/ vj/bug.html.
38. Mark Segal and Ophir Frieder. On-the-fly program modification: Systems for dynamic updating. *IEEE Software*, March 1993.
39. Manuel Serrano. Wide classes. In *European Conference on Object-Oriented Programming.* Springer, 1999.
40. Stephen Slade. *Object-Oriented Common Lisp.* Prentice Hall, Upper Saddle River, NJ 07458, 1998. Chapter 13.
41. Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, 1.01 edition, August 1998. http://www.spec.org/osg/jvm98/.
42. R. Taylor, N. Medvidovic, K. Anderson, E. Whitehead, J. Robbins, K. Nies, P. Oreizy, and D. Dubrow. A component- and message-based architectural style for GUI software. *IEE Transactions on Software Engineering*, June 1996.
43. Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the Memory Management International Workshop.* Springer-Verlag, 1992.

OpenJIT: An Open-Ended, Reflective JIT Compiler Framework for Java

Hirotaka Ogawa[1], Kouya Shimura[2], Satoshi Matsuoka[1],
Fuyuhiko Maruyama[1], Yukihiko Sohda[1], and Yasunori Kimura[2]

[1] Tokyo Institute of Technology
[2] Fujitsu Laboratories Limited

Abstract. OpenJIT is an open-ended, reflective JIT compiler framework for Java being researched and developed in a joint project by Tokyo Inst. Tech. and Fujitsu Ltd. Although in general self-descriptive systems have been studied in various contexts such as reflection and interpreter/compiler bootstrapping, OpenJIT is a first system we know to date that offers a stable, full-fledged Java JIT compiler that plugs into existing monolithic JVMs, and offer competitive performance to JITs typically written in C or C++. This is in contrast to previous work where compilation did not occur in the execution phase, customized VMs being developed ground-up, performance not competing with existing optimizing JIT compilers, and/or only a subset of the Java language being supported. The main contributions of this paper are, 1) we propose an architecture for a reflective JIT compiler on a monolithic VM, and identify the technical challenges as well as the techniques employed, 2) We define an API that adds to the existing JIT compiler APIs in "classic" JVM to allow reflective JITs to be constructed, 3) We show detailed benchmarks of run-time behavior of OpenJIT to demonstrate that, while being competitive with existing JITs the time- and space-overheads of compiler metaobjects that exist in the heap are small and manageable. Being an object-oriented compiler framework, OpenJIT can be configured to be small and portable or fully-fledged optimizing compiler framework in the spirit of SUIF. It is fully JCK compliant, and runs all large Java applications we have tested to date including HotJava. We are currently distributing OpenJIT for free to foster further research into advanced compiler optimization, compile-time reflection, advanced run-time support for languages, as well as other areas such as embedded computing, metacomputing, and ubiquitous computing.

1 Introduction

Programming Languages with high-degree of portability, such as Java, typically employ portable intermediate program representations such as bytecodes, and utilize *Just-In-Time compilers (JITs)*, which compile (parts of) programs into native code at runtime. However, all the Java JITs today as well as those for other languages such as Lisp, Smalltalk, and Self, are monolithically architected without provision for user-level extension. Instead, we claim that JITs could be utilized and exploited more opportunely in the following situations:

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 362–387, 2000.
© Springer-Verlag Berlin Heidelberg 2000

- *Platform-specific optimizations* — Execution platforms could be from embedded systems and hand-held devices to large servers and massive parallel processors (MPPs). There, requirements for optimizations differ considerably, according to particular class of applications that the platform is targeted to execute. JITs could be made to adapt to different platforms if it could be customized in a flexible way.
- *Platform-specific compilations* — Some platforms require assistance of compilers to generate platform-specific codes for execution. For example, DSM (Distributed-Shared Memory) systems and persistent object systems require specific compilations to emit code to detect remote or persistent reference operations. If one were to implement such systems on Java, one not only needs to modify the JVM, but also the JIT compiler. We note that, as far as we know, representative work on Java DSM (cJVM[2]) and persistent objects (PJama[3]) lack JIT compiler support for this very reason.
- *Application-specific optimizations* — One could be more opportunistic by performing optimizations that are specific to a particular application or a data set. This includes techniques such as selection of compilation strategies, runtime partial evaluation, as well as application-specific idiom recognition. By utilizing application-specific as well as run-time information, the compiled code could be made to execute substantially faster, or with less space, etc. compared to traditional, generalized optimizations. Although such techniques have been proposed in the past, it could become a generally-applied scheme and also an exciting research area if efficient and easily customizable JITs were available.
- *Language-extending compilations* — Some work stresses on extending Java for adding new language features and abstractions. Such extensions could be implemented as source-level or byte-code level transformations, but some low-level implementations are very difficult or inefficient to support with such higher-level transformations in Java. The abovementioned DSM is a good example: Some DSMs permit users to add control directives or storage classifiers at a program level to control the memory coherency protocols, and thus such a change must be done at JVM and native code level. One could facilitate this by encoding such extensions in bytecodes or classfile attributes, and customizing the JIT compilers accordingly to understand such extensions.
- *Environment- or Usage-specific compilations and optimizations* — Other environmental or usage factors could be considered during compilation, such as adding profiling code for performance instrumentation, debugging etc. [1]

Moreover, with Java, we would like these customizations to occur within an easy framework of portable, security-checked code downloaded across the network. Just as applets and libraries are downloadable on-the-fly, we would like the JIT compiler customization to be so as well, depending on the specific platform, application, and environment. For example, if a user wants to instrument

[1] In fact we do exactly that in the benchmarking we show later in Section 5, which for the first time characterizes the behavior of a self-descriptive JIT compiler.

his code, he will want to download the (trusted) instrumentation component on-the-fly to customize the generated code accordingly.

Unfortunately, most Java JITs today are architected to be closed and monolithic, and do not facilitate interfaces, frameworks, nor patterns as a means of customization. Moreover, JIT compilers are usually written in C or C++, and live in a completely separate scope from normal Java programs, without enjoying any of the language/systems benefits that Java provides, such as ease of programming and debugging, code safety, portability and mobility, etc. In other words, current Java JIT compilers are "black boxes", being in a sense against the principle of modular, open-ended, portable design ideals that Java itself represents.

In order to resolve such a situation, the collaborative group between Tokyo Institute of Technology and Fujitsu Limited sponsored by Information Promotion Agency of Japan, have been working on a project OpenJIT[19] for almost the past two years. OpenJIT itself is a "reflective" Just-In-Time open compiler framework for Java written almost entirely in Java itself, and plugs into the standard Sun JDK 1.1.x and 1.2 JVMs. All compiler objects coexist in the same heap space as the application objects, and are subject to execution by the same Java machinery, including having to be compiled by itself, and subject to static and dynamic customizations. At the same time, it is a fully-fledged, JCK (Java Compatibility Kit) compliant JIT compiler, able to run production Java code. In fact, as far as we know, it is the ONLY Java JIT compiler whose source code is available in public, and is JCK compliant other than that of Sun's. And, as the benchmarks will show, although being constrained by the limitations of the "classic" JVMs, and still being in development stage lacking sophisticated high-level optimizations, it is nonetheless equal to or superior to the Sun's (classic) JIT compiler on SpecJVM benchmarks, and attains about half the speed of the fastest JIT compilers that are much more complex, closed, and requires a specialized JVM. At the same time, OpenJIT is designed to be a compiler framework in the sense of Stanford SUIF[28], in that it facilitates high-level and low-level program analysis and transformation framework for the users to customize.

OpenJIT is still in active development, and we are distributing it for free for non-commercial purposes from `http://www.openjit.org/`. It has shown to be quite portable, thanks in part to being written in Java—the Sparc version of OpenJIT runs on Solaris, and the x86 version runs on different breeds of Unix including Linux, FreeBSD, and Solaris. We are hoping that it will stem and cultivate interesting and new research in the field of compiler development, reflection, portable code, language design, dynamic optimization, and other areas.

The purpose of the paper is to describe our experiences in building OpenJIT, as well as presenting the following technical contributions:

1. We propose an architecture for a reflective JIT compiler framework on a monolithic "classic" JVM, and identify the technical challenges as well as the techniques employed. The challenges exist for several reasons, that the JIT compiler is reflective, and also the characteristics of Java, such as its pointer-safe execution model, built-in multi-threading, etc.

2. We show an API that adds to the existing JIT compiler APIs in "classic" JVM to allow reflective JITs to be constructed. Although still early in its design, and requiring definitions of higher-level abstractions as well as additional APIs for supporting JITs on more modern VMs, we nonetheless present a minimal set of APIs that were necessary to be added to the Java VM in order to facilitate a Java JIT compiler in Java.
3. We perform extensive analysis of the performance characteristics of OpenJIT, both in terms of execution speed and memory consumption. In fact, as far as we know, there have not been any reports on any self-descriptive JIT compilation performance analysis, nor memory consumption reports for any JIT compilers. In particular, we show that (1) JIT compilation speed does not become a performance issue, especially during the bootstrap process when much of the OpenJIT compiler is run under interpretation, (2) memory consumption of reflective JITs, however, could be problematic due to recursive compilation, especially in embedded situations, (3) that there are effective strategies to solve the problems, which we investigate extensively, and (4) that the solutions do not add significant overhead to overall execution, due to (1). In fact, the self-compilation time of OpenJIT is quite amortizable for real applications.

2 Overview of the OpenJIT Framework

2.1 OpenJIT: The Conceptual Overview

OpenJIT is a JIT compiler written in Java to be executed on "classic" VM systems such as Sun JDK 1.1.x and JDK 1.2.x. OpenJIT allows a given Java code to be portable and maintainable with compiler customization. With standard Java, the portability of Java is effective insofar as the capabilities and features provided by the JVM (Java Virtual Machine); thus, any new features that has to be transparent from the Java source code, but which JVM does not provide, could only be implemented via non-portable means. For example, if one wishes to write a portable parallel application under multi-threaded, shared memory model, then distributed shared memory (DSM) would be required for execution under MPP and cluster platforms. However, JVM itself does not facilitate any DSM functionalities, nor provide software 'hooks' for incorporating the necessary read/write barriers for DSM implementation. As a result, one must modify the JVM, or employ some ad-hoc preprocessor solution, neither of which are satisfactory in terms of portability and/or performance. With OpenJIT, the DSM class library implementor can write a set of compiler metaclasses so that necessary read/write barriers, etc., would be appropriately inserted into critical parts of code.

Also, with OpenJIT, one could incorporate platform-, application-, or usage-specific compilation or optimization. For example, one could perform various numerical optimizations such as loop restructuring, cache blocking, etc. which have been well-studied in Fortran and C, but have not been well adopted into JITs for excessive runtime compilation cost. OpenJIT allows application of such optimizations to critical parts of code in a pinpointed fashion, specified by either

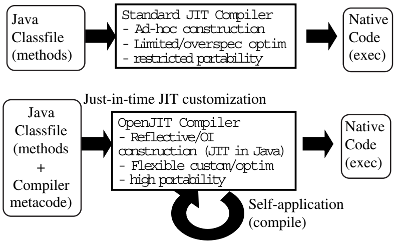


Fig. 1. Comparison of Traditional JITs and OpenJIT

the class-library builder, application writer, or the user of the program. Furthermore, it allows optimizations that are too application and/or domain specific to be incorporated as a general optimization technique for standard compilers, as has been reported by [16].

In this manner, OpenJIT allows a new style of programming for optimizations, portability, and maintainability, compared to traditional JIT compilers, by providing separations of concerns with respect to optimization and code-generation for new features. With traditional JIT compilers, we see in the upper half of Figure 1, the JIT compilers would largely be transparent from the user, and users would have to maintain code which might be too tangled to achieve portability and performance. OpenJIT, on the other hand, will allow the users to write clean code describing the base algorithm and features, and by selecting the appropriate compiler metaclasses, one could achieve optimization while maintaining appropriate separation of concerns. Furthermore, compared to previous open compiler efforts, OpenJIT could achieve better portability and performance, as source code is not necessary, and late binding at run-time allows exploitation of run-time values, as is with run-time code generators.

2.2 Architectural Overview of OpenJIT

The OpenJIT architecture is largely divided into the frontend and the backend processors. The frontend takes the Java bytecodes as input, performs higher-level optimizations involving source-to-source transformations, and passes on the intermediate code to the backend, or outputs the transformed bytecode. The backend is effectively a small JIT compiler in itself, and takes either the bytecode or the intermediate code from the frontend as input, performs lower-level optimizations, and outputs the native code for direct execution. The reason there is a separate frontend and the backend is largely due to modularity and ease of development, especially for higher-level transformations, as well as defaulting to the backend when execution speed is not of premium concern. In particular, we strive for the possibility of the two modules being able to run as independent components.

Upon invocation, the *OpenJIT frontend* system processes the bytecode of the method in the following way: The *decompiler* recovers the AST of the original Java source from the bytecode, by recreating the control-flow graph of the source program. At the same time, the *annotation analysis module* will obtain annotating info on the class file, which will be recorded as attribute info on the AST[2]. Next, the obtained AST will be subject to optimization by the *(higher-level) optimization module*. Based on the AST and control-flow information, we compute the data & control dependency graphs, etc., and perform program transformation in a standard way with modules such as *flowgraph construction module*, *program analysis module*, and *program transformation module*. The result from the OpenJIT frontend will be a new bytecode stream, which would be output to a file for later usage, or an intermediate representation to be used directly by the OpenJIT backend.

The *OpenJIT backend* system, in turn, performs lower-level optimization over the output from the frontend system, or the bytecodes directly, and generates native code. It is in essence a small JIT compiler in itself. Firstly, when invoked as an independent JIT compiler bypassing the frontend, the *low-level IL translator* analyzes and translates the bytecode instruction streams to low-level intermediate code representation using stacks. Otherwise the IL from the frontend is utilized. Then, the *RTL Translator* translates the stack-based code to intermediate code using registers (RTL). Here, the bytecode is analyzed to divide the instruction stream into basic blocks, and by calculating the depth of the stack for each bytecode instruction, the operands are generated with assumption that we have infinite number of registers. Then, the *peephole optimizer* would eliminate redundant instructions from the RTL instruction stream, and finally, the *native code generator* would generate the target code of the CPU, allocating physical registers. Currently, OpenJIT supports the SPARC and the x86 processors as the target, but could be easily ported to other machines. The generated native code will be then invoked by the Java VM, as described earlier.

3 Overview of the OpenJIT Frontend System

As described in Section 2, the OpenJIT frontend system provides a Java class framework for higher-level, abstract analysis, transformation, and specialization of Java programs which had already been compiled by `javac`: (1) The decompiler translates the bytecode into augmented AST, (2) analysis, optimizations, and specialization are performed on the tree, and (3) the AST is converted into the low-level IL of the backend system, or optionally, a stream of bytecodes is generated.

Transformation over AST is done in a similar manner to Stanford SUIF, in that there is a method which traverses the tree and performs update on a node or a subtree when necessary. There are a set of abstract methods that are invoked as a hook. The OpenJIT frontend system, in order to utilize such a hook functionality according to user requirements, extends the class file (albeit in a conformable way so that it is compatible with other Java platforms) by adding annotation info to the classfile. Such an info is called “classfile annotation”.

[2] In the current implementation, the existence of annotation is a prerequisite for frontend processing; otherwise, the frontend is bypassed, and the backend is invoked immediately.

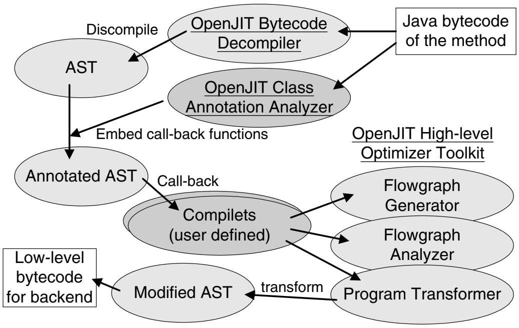

Discompile
OpenJIT Bytecode Decompiler
Java bytecode of the method
AST
OpenJIT Class Annotation Analyzer
Embed call-back functions
OpenJIT High-level Optimizer Toolkit
Annotated AST
Call-back
Flowgraph Generator
Compilets (user defined)
Flowgraph Analyzer
Low-level bytecode for backend
Modified AST
transform
Program Transformer

Fig. 2. Overview of OpenJIT Frontend System

The overall architecture of the OpenJIT frontend system is as illustrated in Fig. 2, and consists of the following four modules:

1. *OpenJIT Bytecode Decompiler*
   Translates the bytecode stream into augmented AST. It utilizes a new algorithm for systematic AST reconstruction using dominator trees.
2. *OpenJIT Class Annotation Analyzer*
   Extracts classfile annotation information, and adds the annotation info onto the AST.
3. *OpenJIT High-level Optimizer Toolkit*
   The toolkit to construct "compilets", which are modules to specialize the OpenJIT frontend for performing customized compilation and optimizations.
4. *Abstract Syntax Tree Package*
   Provides construction of the AST as well as rewrite utilities.

For brevity, we omit the details of the frontend system. Interested readers are referred to [20].

4 OpenJIT—Backend and Its Technical Issues

4.1 Overview of the OpenJIT Backend System

As a JIT compiler, the high-level overview of the workings of OpenJIT backend is standard. The heart of the low-level IL translator is the `parseBytecode()` method of the `ParseBytecode` class, which parses the bytecode and produces an IL stream. The IL we defined is basically an RISC-based, 3-operand instruction set, but is tailored for high affinity with direct translation of Java instructions into IL instruction set with stack manipulations for later optimizations. There are 36 IL instructions, to which each bytecode is translated into possibly a sequence of these instructions. Some complex instructions are translated into calls into run-time routines. We note that the IL translator is only executed when the OpenJIT backend is used in a standalone fashion; when used in conjunction with the frontend, the frontend directly emits IL code of the backend.

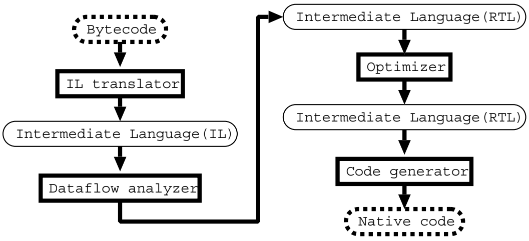

Bytecode
IL translator
Intermediate Language(IL)
Dataflow analyzer
Intermediate Language(RTL)
Optimizer
Intermediate Language(RTL)
Code generator
Native code

Fig. 3. Overview of the OpenJIT Backend System

Then, RTL converter translates the stack-based IL code to register based RTL code. The same IL is used, but the code is restructured to be register-based rather than encoded stack operations. Here, a dataflow analyzer is then run to determine the type and the offset of the stack operands. We assume that there are infinite number of registers in this process. In practice, we have found that 24–32 registers are sufficient for executing large Java code without spills when no aggressive optimizations are performed[24]. Then, the *peephole optimizer* would eliminate redundant instructions from the RTL instruction stream.

Finally, the *native code generator* would generate the target code of the CPU. It first converts IL restricting the number of registers, inserting appropriate spill code. Then the IL sequence is translated into native code sequence, and ISA-specific peephole optimizations are performed. Currently, OpenJIT supports the SPARC and x86 processors as the target, but could be easily ported to other machines[3]. The generated native code will be then invoked by the Java VM, upon which the *OpenJIT runtime module* will be called in a supplemental way, mostly to handle Java-level exceptions.

The architectural outline of the OpenJIT backend is illustrated in Figure 3. Further details of the backend system can be found in [23].

4.2 Technical Challenges in a Reflective Java JIT Compiler

As most of OpenJIT is written in Java, the bytecode of OpenJIT will be initially interpreted by the JVM, and gradually become compiled for faster, optimized execution. Although this allows the JIT compiler itself to adapt to the particular execution environment the JIT optimizes for, it could possibly give rise to the following set of problems:

1. *Invoking the Java-based JIT compiler from within the JVM*
   As the JIT compiler is invoked in the midst of a call chain of the base Java program. There must be a smooth way to massaging the JVM into invoking a JIT compiler in Java in a separate context.

[3] Our experience has been that it has not been too difficult to port from SPARC to x86, save for its slight peculiarities and small number of registers, due in part being able to program in Java. We expect that porting amongst RISC processors to be quite easy.

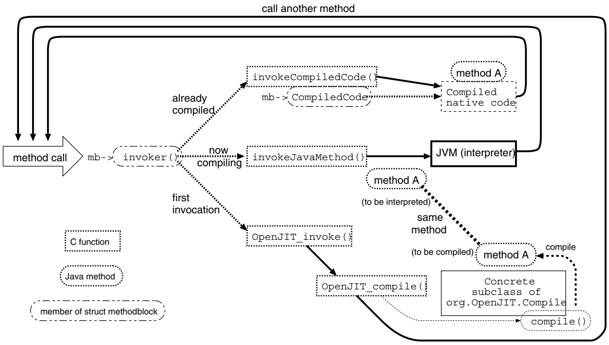


Fig. 4. Invoking OpenJIT

2. *Recursive Compilation*
   The current OpenJIT is designed to be *entirely* bootstrapped in “cold” mode, i.e., no parts of the JIT compiler are precompiled. Thus, as is with any reflective system, there must be some mechanism to stop the infinite recursive process, and “bottom out”. This is a little more subtle than conventional compiler bootstrapping, as compilation occurs at runtime coexisting with compilation of applications; furthermore, the mechanism must be safe w.r.t. Java multi-threading, i.e., no deadlocks should occur.
3. *Speed and Memory Efficiency of the JIT compiler*
   A JIT compiler is beneficial only if the combined (compilation time + execution time) is smaller than the interpretation time under JVM. In more practical terms, OpenJIT must compete with traditional C-based JIT compilers for performance. Here, because of the interpretation and possible slowness of JIT execution even if itself were JIT compiled due to quality of generated native code, it is not clear if such goals could be satisfied. Moreover, memory efficiency is of primary concern, especially for embedded systems. In this regard, there is a particular issue not present in C-based JIT compilers.
4. *Lack of appropriate API for Java-written JIT compilers in standard JVM*
   A JIT compiler must be able to introspect and modify various data structures within the JVM. Unfortunately, JVM does not have any APIs for that purpose, primarily because it is likely that JIT compilers were assumed to be written with a low-level language such as C. For this purpose, there must be appropriate Java-level APIs which must be reasonably portable for JVM introspection in OpenJIT.

Again, for brevity, we only cover the most salient technical features here: for complete technical details readers are referred to [25].

Invoking the Java-based JIT compiler from within the JVM. In a “classic” JVM, for each method, both JIT compilation and transfer of control to the native method happens at the point of the subject method invocation. The JVM interpreter loop is structured as follows. When a method is invoked, the invoker function of the methodblock structure (a structure internal to the JVM which embodies various info pertaining to a particular method) mb is called. Under interpretive execution, this in turn calls the JVM to generate a new Java stack frame. The first argument of invoker() function o is the class object for static method calls, and the invoked object on normal method calls. The second argument mb is a pointer to the methodblock structure, etc.

```
while(1) {
    get opcode from pc
    switch(opcode) {
    ...(various implementation of the JVM bytecodes)
    callmethod:
        mb->invoker(o, mb, args_size, ee);
        frame = ee->current_frame; /* setup java frame */
        pc = frame->lastpc; /* setup pc*/
        break;
    }
}
```

As showed in Figure 4, we substitute the value of the invoker in methodblock structure of every method to OpenJIT_invoke when a class is loaded. The OpenJIT_invoke function is defined as follows in C:

```
    bool_t OpenJIT_invoke(JHandle *o, struct methodblock *mb,
                          int args_size, ExecEnv *ee)
```

This function in turns calls the OpenJIT_compile() in the C runtime to dynamically compile the method. Thereafter, the control is transferred to mb->invoker, transferring control to the just compiled method. The called OpenJIT_compile() performs the following functions:

1. *Mutual exclusion to prevent simultaneous compilation of the same method* — We must prevent multiple threads from compiling the same method at the same time with proper mutual execution using a compile lock. We reserve a bit in the methodblock structure as a lock bit.
2. *Setup of invoker and CompiledCode fields in the methodblock structure* — When a method is invoked, and subject to compilation, we reset the invoker and other fields in the methodblock so that any subsequent invocation of the method will have the method run by the interpreter during compilation. This allows natural handling of recursive self-compilation of OpenJIT compiler classes.
3. *Invocation of the body of the JIT compiler* — The Java method to invoke the compiler is then upcalled. An instance of a new JIT compiler in Java (to be more specific, its upcall entry class) is allocated and initialized for each JIT compiler invocation. Then, the compile() method of the instantiated entry class is up with do_execute_java_method_vararg(). Note that the current call context is preserved in the stack; that is to say, the same thread is utilized to make the upcall.

4. *Postprocessing of JIT compilation* — After compilation, control returns to the C runtime. At this point, most of the compiler becomes garbage, except for the persistent information that must be maintained across method compilations. This is to facilitate dynamic change in the compiler with compilets, and also to preserve space, directly exploiting the memory management feature of Java. If the compilation is successful, we set the invoker field of the methodblock structure to the compiled native code. When compilation fails: The methodblock field values are restored to their original values. [4]

In this manner, the JIT compiler in Java is smoothly invoked on the same execution thread. In practice it is much more complicated, however, due to possibility of exceptions, JIT compilation occurring even on calls from native methods, advanced features such as backpatching, inlining, and adaptive compilation. Some of the issues are further discussed below, while for the rest refer to [25].

Recursive Compilation. Recursive compilation is handled at the C runtime level of OpenJIT with simple locking mechanism, as we see in the following simplified code fragment (in practice, it would include more code such as support for adaptive compilation):

```
COMPILE_LOCK(ee);
if (COMPILE_ON_THE_WAY(mb)) {
    /* now compiling this method. avoid from double compiling */
    COMPILE_UNLOCK(ee);
    return;
}
START_COMPILE(mb);
/* reset invoker temporarily */
mb->invoker = (mb->fb.access & ACC_SYNCHRONIZED) ?
    invokeSynchronizedJavaMethod : invokeJavaMethod;
/* reset dispatcher temporarily */
mb->CompiledCode = (void *)dispatchJVM;
COMPILE_UNLOCK(ee);
```

This is essentially where the compilation “bottoms out”; once the method starts to be compiled, a lock is set, and further execution of the method will be interpreted. In fact, in Java we actually obtain this behavior *for free*, as mutual exclusion of multi-threaded compilation has to be dealt with in any case, defaulting to interpretation.

However, in the case of recursive compilation, there are some issues which do not exist for C-based JIT compilers:

[4] In practice, the invoker field is not directly substituted for the compiled native method, but rather we invoke a native code stub, depending on the type of the return argument. This is done to handle exceptions, java reflection, calls between native and interpreted code, etc.

- *Possibility of Deadlocks* — We must be assured that, as long as JIT compiler obeys the locking protocol, recursive multi-threaded compilation does not cause any deadlocking situations. This is proven by showing that cyclic resource dependencies will not exist between the multi-threaded compilations. Let the dependencies between the methods be denoted $m_1^c \longrightarrow m_2^c$, where for execution of compiled method m_1^c we need to execute a compiled method m_2^c. We further distinguish compiled and interpreted execution of methods with m^c and m^i, respectively. Then, starting from the entry method as a node, graph of dependency relations will clearly form a tree for single-threaded case. For multi-threaded case, however, it must be shown that arbitrary interleavings of the tree via possible self compilation will only create DAGs. Informally this simply holds because all m_i's will not be dependent on any other nodes, and thus the cycle will have to be formed amongst m_c's, which is not possible.

  We also note that, in practice, deadlocks could and does occur not only between the JIT compiler and the JVM. One nasty bug which took a month to discover was in fact such a deadlock bug. As it turns out, the "classic" JVM locks the constant pool for a class when its finalizer is run. This could happen just when OpenJIT tries to compile the finalizer method, resulting in a deadlock.
- *Speed and Memory Performance Problems* — Aside from the JIT compiler merely working, we must show that the JIT compiler in Java could be time and memory efficient. The issue could be subdivided into cases where the OpenJIT is compiling (1) application methods, and (2) OpenJIT methods. The former is simply shown by extensive analysis of standard benchmarks in Section 5, where it is shown that OpenJIT achieves good time and memory performance and despite being constrained by the limitations of the "classic" VM, such as handle-based memory systems implementation, non-strict and non-compacting GC, slow monitor locking, etc. The latter is much more subtle: because of recursive compilation, two undesirable phenomena occur. (A) compilation of a single application bytecode will set off a chain of recursive compilations, due to the dependency just discussed. This has the effect of accumulating compiling contexts of almost the entire OpenJIT system, putting excessive pressure on the memory system. (B) We could prevent the situation by employing adaptive compilation and defaulting back to interpretation earlier, but this will have the effect of slowing down the bootstrap time, as long as possibly having some residual effect on application compilation due to some OpenJIT compiler methods still being interpreted. (A) and (B) are strongly interrelated; in the worst case, we will be trading speed, especially the bootup time, for space. On the other hand, one could argue that little penalty is incurred by adaptive means, not because of the typical execution frequency argument, but rather, that because of recursive compilation, much of the OpenJIT system could be compiled under interpretation in the first place. We perform extensive performance analysis to investigate this issue in Section 5.

Lack of appropriate API for Java-written JIT compilers in standard JVM. None of the current Java VMs, including the "classic" VM for which OpenJIT is implemented, have sufficient APIs for implementing a JIT compiler in Java. In particular, JVM basically only provides APIs to *invoke* a C-based JIT compiler, but does not provide sufficient APIs for generalized introspection or intersession features. Note that we cannot employ the Java reflection API either, for it abstracts out the information required by the JIT compiler.

Instead, we define a set of native methods as a part of the OpenJIT runtime. The `Compile` class declares the following native methods, which are defined in `api.c` of the distribution. There are 17 methods in all, which can be categorized as follows:

- *Constant pool introspection methods*

```
public final native int ConstantPoolValue(int index)
private final native int ConstantPoolTypeTable(int index)
public final int ConstantPoolType(int index)
public final boolean ConstantPoolTypeResolved(int index)
public final String ConstantPoolClass(int index)
private native byte[] ConstantPoolClass0(int index)
public final String ConstantPoolName(int index)
private native byte[] ConstantPoolName0(int index)
public final native int ConstantPoolAccess(int index)
public final native byte[] ConstantPoolMethodDescriptor(int index)
public final native int ConstantPoolFieldOffset(int index)
public final native int ConstantPoolFieldAddress(int index)
```

- *Native method allocation and reflection*

```
public final native void NativeCodeAlloc(int size)
public final native int NativeCodeReAlloc(int size)
public final native void setNativeCode(int pc, int code)
public final native int getNativeCode(int pc)
private native byte[] MethodName()
```

- *Class resolution methods (used for inlining)*

```
public final native void initParser(int caller_cp, int index)
public final native void resolveClass(int caller_cp, int index)
```

As one can see, majority of the methods are such that either introspective or intercessive operations being performed on the JVM.

The current API is sufficient, but admittedly too low level of abstraction, in that it exposes too much of the underlying VM design; indeed, our goal is to allow JITs to be a customizable and portable hook to the Java system, and thus, have OpenJIT be portable across different kinds of VMs. For this purpose, in the next version of OpenJIT, we plan to design a substantially higher-level API, abstracting out the requirements of the different VMs. The implementation of the API for "classic" VM will sit on the current APIs, but other VMs will have different implementations of native methods.

Another issue is the safety of the API. In the current implementation, the OpenJIT native method APIs are accessible to all the classes, including the application classes. It is easy to restrict the access to just the compiler classes (those with path `org.OpenJIT.`), but this will preclude user-defined compilets. Some form of security/safety measures with scope control, such as restricting access only to signed classfiles, might be necessary. We are currently investigating this possibility to utilize the security API in JDK 1.2.

5 Performance Analysis of OpenJIT

We now analyze the behavior of OpenJIT with detailed benchmarks. As mentioned earlier, our concern is both execution speed and memory usage. The former is obvious, as the execution overhead of the JIT compiler itself as well as quality of generated code will have to match that of conventional JIT compilers. Memory usage is also important, especially in areas such as embedded computing, one of major Java targets.

All the OpenJIT objects, except for the small C runtime system, coexists in the heap with the target application. The necessary working space includes that of various intermediate structures of compiler metaobjects that the OpenJIT builds, including various flowgraphs, intermediate code, etc., and persistent data, such as the resulting native code. Standard C-based JIT compilers will have to allocate such structures outside the Java heap; thus, memory usage is fragmented, and efficient memory management of the underlying JVM is not utilized. For OpenJIT, since both the application and compiler metaobjects will coexist in the heap, it might seem that we would obtain the most efficient usage of heap space.

On the other hand, the use of Java objects, along with automated garbage collection, could be less memory efficient than C-based JITs. Moreover as mentioned in Section 4, there could be a chain of recursive compilations which will accumulate multiple compilation contexts, using up memory. It is not clear what kind of adaptive compilation techniques could be effective in decreasing the accumulation, while not resulting in substantial execution penalty.

5.1 Benchmarking Environment

As an Evaluation Environment, we employed the following platform, and pitted OpenJIT against Sun's original JIT compiler (sunwjit) on JDK 1.2.2 (ClassicVM).

- Sun Ultra60 (UltraSparc II 300MHz×2, 256MB)
- Solaris 2.6-J
- JDK 1.2.2 (ClassicVM)

We took six programs from the SPECjvm98 benchmark, as well as the simple "Hello World" benchmark. The six—_201_compress (file compression), _202_jess (expert system), _209_db (DBMS simulator), _213_javac (JDK 1.0.2 compiler),

Table 1. Code size of OpenJIT and C runtimes

	classes (files)	methods	# lines	classfile (stripped binary) bytes
Frontend	243	1,439	24,148	629,062
Backend (sparc)	23	182	7,560	118,592
Backend (x86)	21	182	8,085	118,125
C runtime(sparc)	3		3,565	42,556
C runtime(x86)	3		3,752	28,928
sunwjit (sparc)				234,112
sunwjit (x86)				146,508

and _227_mtrt (multi-threaded raytracer), and _228_jack (parser generator)—have been chosen as they are relatively compute intensive, do not involve mere simple method call loops, and not reliant on runtime native calls such as networks, graphics, etc. “Hello World” benchmark superficially only makes a call to System.out.println(), but actually it will have executed almost the entire OpenJIT system, the Java packages that OpenJIT employs, as well as the constructors of system classes. This allows us to observe the bootstrap overhead of the OpenJIT system.

In order to obtain the precise profile information for memory allocation, we employed the JVMPI (Java Virtual Machine Profiler Interface) of JDK 1.2.2. Additionally, we extended OpenJIT to output its own profile information. This is because it is difficult to determine with JVMPI whether the allocated compiler metaobject is being used to compile application methods, or used for recursive compilation, because JVMPI merely reports both to be of the same class (say, merely as instances of `ILnode`, etc.). By combining JVMPI and OpenJIT profile information, we obtain precise information of how much space the live OpenJIT compiler metaobjects occupy, how much native code is being generated, how much of the native code is that of OpenJIT, along the execution timeline. Also, how much classfiles are being loaded, how many methods are being compiled, and what is the percentage of the OpenJIT classes, can be profiled as well.

Such profiling is done in real-time, in contrast to the simulation based profiling of SpecJVM memory behavior in [6]. Such an approach is difficult to apply for our purpose, as JIT compilation is being directly involved, resulting in code not directly profilable with JVM simulation. Our compiler-assisted profiling allows us to obtain almost as precise an information as that of [6] at a fraction of time. Nevertheless, the profile information generated is quite large, reaching several hundred megabytes for each SpecJVM run.

5.2 Benchmarking Contents

The Size of OpenJIT. We first show the code size of OpenJIT compared to sunwjit. As we can see, the frontend is approximately 3 times the size of backend in terms of number of lines, and factor of approximately 8–10 larger in terms of number of classes and methods. This is because the frontend contains numerous small classes representing syntactic entities of Java, whereas the backend has much larger method size, and the backend IL does not assign a class for each instruction. We also see that the combined size of OpenJIT backend and C runtime is smaller than sunwjit, but when it is self-compiled, the x86 version

Table 2. Baseline Performance

Program	JIT	class#	alloc obj#	allocsize[MB]	GC#	time
			(openjit)	(openjit)		
Hello	interpreter	167	4,890(—)	0.273(—)	0	0.380
	sunwjit	172	5,244(—)	0.285(—)	0	0.450
	openjit	185	90,600(74,831)	2.906(2.316)	5	1.270
	openjit-int	185	37,059(31,149)	1.265(0.941)	2	1.280
_201_ compress	interpreter	224	15,547(—)	110.640(—)	20	673.910
	sunwjit	226	9,399(—)	110.266(—)	16	89.620
	openjit	241	136,328(107,197)	114.330(3.318)	21	72.530
	openjit-int	241	81,910(62,742)	112.662(1.918)	18	74.460
_202_ jess	interpreter	373	7,951,562(—)	221.919(—)	547	148.550
	sunwjit	375	7,936,214(—)	221.190(—)	565	65.750
	openjit	390	8,103,973(142,626)	226.383(4.402)	528	62.530
	openjit-int	390	8,049,403(98,045)	224.710(2.998)	532	62.160
_209_ db	interpreter	218	3,218,293(—)	63.249(—)	33	307.480
	sunwjit	220	3,213,851(—)	63.095(—)	32	142.160
	openjit	235	3,343,820(109,778)	67.104(3.398)	39	172.830
	openjit-int	235	3,289,249(65,197)	65.431(1.994)	37	182.080
_213_ javac	interpreter	386	5,972,713(—)	147.288(—)	80	200.940
	sunwjit	388	5,936,663(—)	145.458(—)	69	94.850
	openjit	403	6,181,295(208,562)	154.486(6.478)	77	102.960
	openjit-int	403	6,126,571(164,145)	151.531(5.080)	67	108.850
_227_ mtrt	interpreter	239	6,382,222(—)	84.118(—)	90	173.510
	sunwjit	241	6,376,266(—)	83.902(—)	90	59.430
	openjit	256	6,524,115(124,549)	88.467(3.855)	96	56.640
	openjit-int	256	6,469,545(79,968)	86.794(2.451)	93	56.980
_228_ jack	interpreter	270	6,878,777(—)	150.755(—)	451	196.330
	sunwjit	272	6,868,951(—)	150.353(—)	465	66.669
	openjit	287	7,046,695(152,625)	155.818(4.707)	286	66.970
	openjit-int	287	6,992,109(108,001)	154.144(3.302)	276	68.010

could get larger. Thus, this raises an interesting issue of what happens if we run the compiler always interpreted in embedded situations; in the subsequent benchmark, we will also investigate this possibility.

Baseline Performance. We next observe the baseline execution time and memory usage characteristics of OpenJIT. We set the heap limit to 32MBytes (as mandated by the SpecJVM98 benchmarks) comparing the execution of JVM interpreter, sunwjit, OpenJIT with self compilation, and OpenJIT without self compilation. Table 2 shows for each execution, how many classes are loaded and their sizes, how many objects are allocated (parenthesis indicates how many OpenJIT compiler metaobjects), how much memory size are allocated (and that of OpenJIT compiler metaobjects), wallclock execution time, and number of GCs.

Figure 5 additionally show consumed overall heap space, live OpenJIT object heap space, along the time axis. This shows the process of compiler bootstrapping. The compilation in OpenJIT was set to be most aggressive i.e., all the methods are JIT compiled on their first invocations, and the entire frontend had been turned off, and are not loaded.

The Hello benchmark exemplifies the overhead of bootstrapping openjit and openjit-int; compared to sunwjit, we see approximately 2.8 times increase in startup time, indicating that compilation with OpenJIT incurs approximately ×3 overhead over sunwjit. On the other hand, difference between openjit and

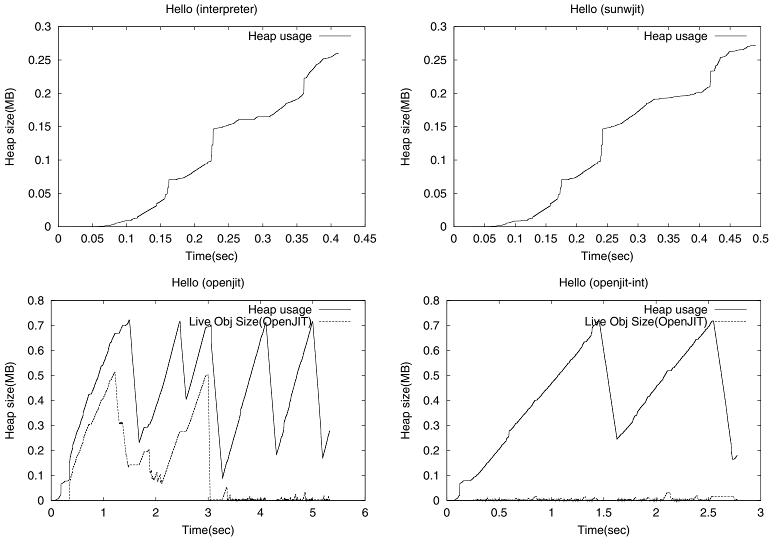


Fig. 5. Timeline behavior of heap usage and live object allocated by OpenJIT (interpreter, sunwjit, openjit, openjit-int). The measured time of this figure contains the overhead of profiling, so it did not exactly match the results of Table 2.

openjit-int is negligibly small; this indicates that overhead of self compilation is almost negligible, but rather, the overhead of system and library classes are substantial (we observe approximately 457 methods compiled, as opposed to 128 methods for OpenJIT).

For the six SPECjvm98 benchmarks, we see that the overhead is well amortized, and OpenJIT is competitive with sunwjit, sometimes superior. The running time of programs range between 56–172 seconds, so the overhead of JIT compilation is well amortized, even for openjit-int, given the relative expense of OpenJIT compilation over sunwjit. Moreover, since method-specific openjit compiler metaobjects are mostly thrown away on each compilation, in principle we do not occupy memory compared to sunwjit (Fig. 5) In fact, we may be utilizing memory better due to sharing of the heap space with the application. The runtime comparison of execution times of each program depends on each program. For compress, openjit was 20% superior, whereas sunwjit was faster by about 18%. Other benchmarks are quite similar in performance. Even small but unnegligible difference in compilation overhead, OpenJIT is likely producing slightly superior code on average.

We do observe some anomalies, however. Firstly, for most cases OpenJIT had increased invocations of GCs due to heap coexistence; but for jack and jess, OpenJIT had less GC invocations, by approx. 40% and 5%, respectively. This is attributable to unpredictable behavior of conservative GC in the "classic" JVM; it is likely that by chance, the collector happened to mistake scalars for pointers on the stack. Neither really contributes significantly to performance differences. Another anomaly is that, in many cases openjit-int was faster than openjit with self-compilation. This somewhat contradicts our observation that compilation DOES incur some overhead, as difference between interpreted and compiled executions of OpenJIT itself should manifest, but doesn't.

Figure 5 shows the timeline track of the amount of heap usage by the entire program, OpenJIT (openjit) and interpreted OpenJIT (openjit-int), respectively, for the Hello benchmark. Again, we observe that during bootstrapping, openjit and openjit-int require approximately 700Kbytes of heap space, which is about 2.6 times the heap space as sunwjit and pure interpreter. Since openjit-int does not allocate metaobjects to compile itself, and the amount being consumed to compile methods of other classes are small, we attribute the consumption to the system objects with the libraries being called from OpenJIT, and immediately released.

The Hello benchmark also verifies that there are two phases of execution for OpenJIT. Firstly, there is a bootstrap phase where the entire OpenJIT is aggressively compiled, accumulating multiple compilation contexts in the call

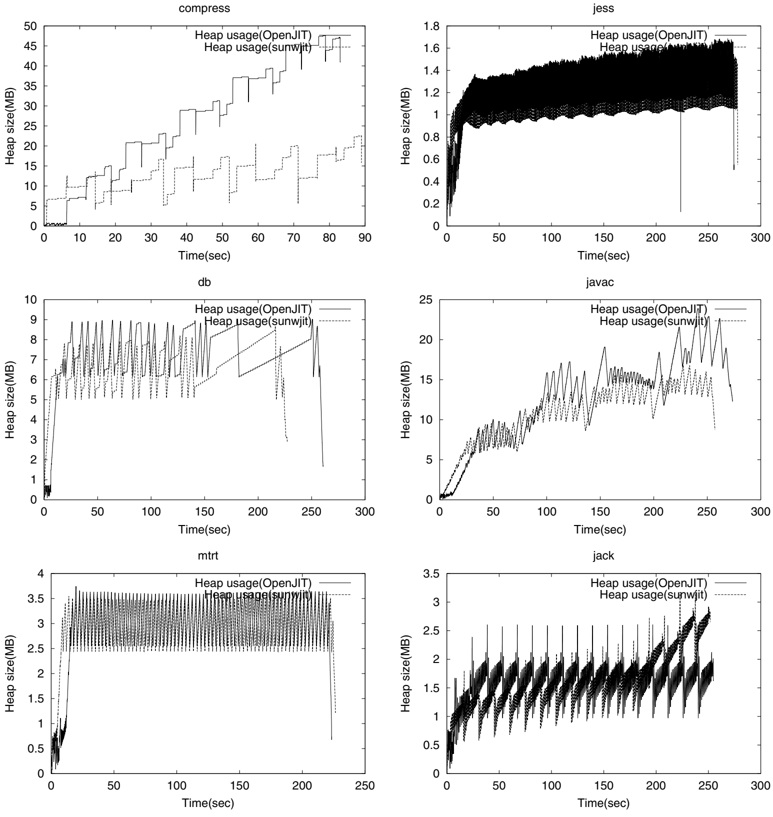


Fig. 6. Timeline behavior of heap usage with SPECjvm98 (sunwjit vs. openjit). The measured time of this figure contains the overhead of profiling, so it did not exactly match the results of Table 2.

Table 3. SPECjvm98 results on Linux (x86)

Benchmark	OpenJIT	interpreter	sunwjit	IBM1.1.8
_200_check	0.043	0.082	0.106	0.042
_201_compress	40.828	316.687	59.338	18.236
_202_jess	29.783	90.434	48.493	13.142
_209_db	85.881	203.289	119.228	40.259
_213_javac	56.227	120.199	70.698	30.964
_222_mpegaudio	40.911	263.870	41.705	11.942
_227_mtrt	30.101	96.156	37.337	17.014
_228_jack	28.403	107.049	49.176	10.751

chain of the JIT compiler. Thus, the space required is proportional to the critical path in the call chain. Then, it quickly falls off, and transcends into a stable phase where most parts of OpenJIT have been compiled, and only application methods are being compiled and executed.

No matter how the memory is being used, the amount of additional heap space required for recursively compiling OpenJIT will not be a problem for modern desktop environments, in some situations. A typical desktop applications consumes orders of magnitude more space: for example, our measurements in Figure 6 for `javac` shows it consumes more than 20 Mbytes[5]. However, for embedded applications, such an overhead might be prohibitive. As discussed earlier, this could be suppressed using less aggressive, adaptive compilation similar to the Self compiler[12], but it is not clear what strategy will achieve good suppression while not sacrificing performance. In the next section we consider several adaptive strategies for suppression.

We have also taken some benchmarks on the x86 version of OpenJIT, and compared it against IBM's JDK 1.1.8 JIT compiler, which is reputed to be the fastest JIT compiler for x86, in neck to neck with Sun's Hotspot. Table 3 shows the results: we see that, for most benchmarks, OpenJIT x86 is superior to sunwjit, and runs about half the speed of IBMs JIT, despite being constrained by the "classic" JVM.

5.3 Adaptive OpenJIT Compilation Strategies

There are several criteria in the design space for adaptive compilation in OpenJIT for memory suppression of the bootstrapping phase.

1. *Alteration of JIT compilation frequency* — The most aggressive strategy will compile each method on its first invocation. We reduce the frequency of compilation using the following strategies, with p as a parameter ($p = 2, 4, 8, 16$)
   - JIT compile on pth invocation, deferring to interpretation for the first $p - 1$ invocations (*constant delay*).
   - Assign each method a random number between $[0, p - 1]$, and compile when the number of invocation reaches that number (*random delay*).

[5] [6] reports that with exact GC, the actual usage is approximately 6MBytes. The difference is likely to be an artifact of conservative GC, our close examination has shown.

  - Compile with probability $1/p$ on each invocation (*probabilistic*). reduced probability increased the execution time by

2. *Restriction of methods subject to adaptation* — We could delay compilation for all the methods, or alternatively, only those of the OpenJIT compiler metaclasses. The former obviously will likely consume less space, but the former may be sufficient and/or desirable, as it will not slow down the application itself. We verify this by comparing altering compilation frequency changes to all classes, versus only altering the frequency of OpenJIT method. For the latter, all other methods are compiled on first invocation except for class initializers, which are interpreted.
3. *Restriction of number of simultaneous compilations* — We put global restriction on how many compilations can occur simultaneously. This can be done safely without causing deadlocks. Attempt to compile exceeding this limit will default back to the interpreter. ($L = 1, 8$). Note that, although simultaneous compilation could occur for application methods under multithreading, this primarily restricts the simultaneous occurrence of deep recursive compilation chains on bootstrapping.
4. *Restricting compilation of* `org.OpenJIT.ParseBytecode.parseBytecode()` — This is a special case, as preliminary benchmarks indicated that `parseBytecode()`, is quite large for a single method, (1576 lines of source code, 6861 JVM bytecodes), and thus single compilation of this method creates a large structure in the heap space once it is subject to compilation, irrespective of the strategies used. In order to eliminate the effect, we test cases where compilation of `parseBytecode()` is restricted. In the next version of OpenJIT we plan to factor the method into smaller pieces.

According to the Hello benchmark, in when adaptaion is applied to all the methods, combinations of other schemes effectively yielded reduction in the number and size of objects that are allocated during bootstrapping, without significant increase in bootstrap time. On the other hand, restricting compilation of OpenJIT method only did not yield significant results, except for the case when the entire OpenJIT was interpreted, or when `parseBytecode()` was restricted, again, without significant loss of performance.

The table only shows the *total* memory allocated. In order to characterize the *peak* memory behavior, we present the timeline behavior in Figure 7. Here, for each scheme, the parameter with *lowest* peak is presented. We observe that, (1) probabilistically lowering the frequency helps reduce the peak usage, and (2) `parseBytecode()` dominates the peak. We are currently conducting futher analysis, but it is conclusive that naive frequency adjustment does not help to reduce the peak; rather, the best strategy seems to be to estimate the heap usage based on bytecode length, and supressing compilation once a prescribed limit is exceeded.

Table 4. Alteration of JIT compilation frequency for all methods

criteria	Hello					
	param	method# (openjit)	alloc obj# (openjit)	alloc size [MB] (openjit)	GC#	time
always	–	457(128)	90,594(74,831)	2.905(2.316)	5	1.270
constant delay	2	225(126)	64,827(52,194)	2.117(1.629)	3	1.190
	4	180(116)	57,568(46,133)	1.898(1.445)	3	1.280
	8	161(114)	55,743(44,470)	1.843(1.395)	4	1.250
	16	151(113)	54,069(43,064)	1.793(1.352)	4	1.810
random delay	2	412(127)	84,739(69,746)	2.729(2.162)	5	1.270
	4	301(123)	72,686(59,272)	2.357(1.843)	4	1.190
	8	231(120)	62,628(50,589)	2.050(1.578)	4	1.240
	16	196(115)	61,923(49,790)	2.031(1.558)	3	1.350
probability	2	170(115)	56,383(45,041)	1.862(1.412)	4	1.280
	4	190(115)	58,493(46,877)	1.924(1.466)	4	1.900
	8	149(115)	53,710(42,735)	1.780(1.341)	3	2.010
	16	112(99)	27,066(21,399)	0.955(0.664)	1	0.920

Table 5. Alteration of JIT compilation frequency for OpenJIT methods only

criteria	Hello					
	param	method# (openjit)	alloc obj# (openjit)	alloc size [MB] (openjit)	GC#	time
always	–	457(128)	90,594(74,831)	2.905(2.316)	5	1.270
constant delay	2	457(128)	90,526(74,757)	2.904(2.314)	5	1.240
	4	451(122)	89,616(73,988)	2.877(2.291)	5	1.260
	8	449(120)	88,535(73,179)	2.845(2.265)	5	1.280
	16	446(117)	85,699(71,112)	2.760(2.200)	5	1.800
random delay	2	457(128)	90,534(74,765)	2.904(2.314)	5	1.320
	4	455(126)	90,236(74,491)	2.895(2.306)	5	1.220
	8	452(123)	89,780(74,115)	2.882(2.295)	5	1.230
	16	450(121)	89,299(73,763)	2.868(2.284)	5	1.330
probability	2	450(121)	89,490(73,884)	2.873(2.288)	5	1.270
	4	429(116)	83,831(69,523)	2.703(2.152)	4	1.980
	8	446(117)	85,722(71,136)	2.760(2.201)	5	2.020
	16	441(112)	84,637(70,295)	2.728(2.117)	4	0.910
limit simultaneity	1	457(128)	90,590(74,821)	2.906(2.316)	4	2.530
	8	457(128)	90,514(74,751)	2.903(2.314)	4	1.410
no parseBytecode	–	456(127)	69,463(58,430)	2.248(1.788)	3	1.190
openjit-int	–	329(0)	37,059(31,149)	1.265(0.941)	2	1.280

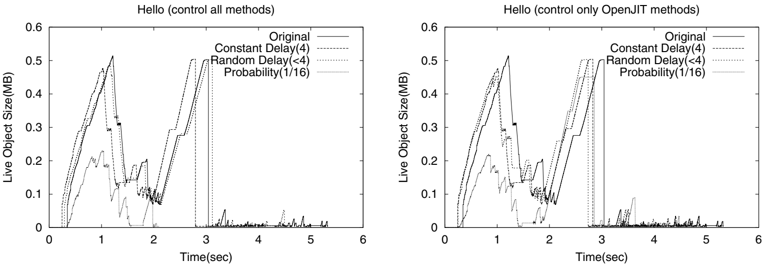


Fig. 7. Timeline behavior of heap usage for adaptive compilation (best cases).

6 Related Work

As mentioned earlier, most modern compilers and language systems are bootstrapped in a self-descriptive fashion, but they do not coexist at runtime. In fact, although Lisp and Smalltalk systems embodied their own compilers written in terms of itself and executable at run-time, they are typically source-to-bytecode compilers, and not bytecode-to-native code compilers, which JITs are. In fact, as far as we know, there have not been any reports of a JIT compiler for a particular language being reflective. Most JIT compilers we have investigated, including those for Lisp, Smalltalk, Java as well as experimental languages such as SELF, have been written in C/C++ or in assembly language.

More recent efforts in self-descriptive, practical object-oriented system is Squeak[14]. Squeak employs the Bluebook[9] self-definition of Smalltalk, then bootstraps it using C, then further optimizes the generated VM. Bootstrapping in Squeak involves the VM only, and not the JIT compiler. The recently-announced JIT Squeak compiler is written in C and basically only merges the code fragments corresponding to individual bytecode. Thus, this is not a true compiler in a sense, but rather a simple bytecode to binary translator. This was done to achieve very quick porting of Squeak to various platforms, and stems from some of the earlier work done in [21].

We know of only two other related efforts paralleling our research, namely MetaXa[10] and Jalapeño[1]. Metaxa is a comprehensive Java reflective system whereby many language features could be reified, including method invocations, variable access, and locking. MetaXa has built its own VM and a JIT compiler; as far as we have communicated with the MetaXa group, their JIT compiler is not full-fledged, and is specific to their own reflective JVM. Moreover, their JIT is reported not robust enough to compile itself.

Jalapeño[1] is a major IBM effort in implementing a self-descriptive Java system. In fact, Jalapeño is an aggressive effort in building not only the JIT compiler, but the entire JVM in Java. The fundamental difference stems from the fact that Jalapeño rests on *its own customized JVM with completely shared address space*, much the same way the C-based JIT compilers are with C-based JVMs. Thus, there is little notion of separation of the JIT compiler and the VM for achieving portability, and the required definition of clean APIs, which is mandated for OpenJIT. For example, the JIT compilers in Jalapeño can access the internal objects of the JVM freely, whereas this is not possible with OpenJIT. So, although OpenJIT did not face the challenges of JVM bootstrapping, this gave rise to investigation of an effective and efficient way of interfacing with a monolithic, existing JVMs, resulting in very different technical issues as have been described in Section 4.

The manner in which Jalapeño bootstraps is very similar to Squeak and other past systems. The way the type safety of Java is circumvented, however, is similar to the technique employed in OpenJIT: there is a class called `Magic`, which defines a set of native methods that implements operations where direct access to VM internals are required. In OpenJIT, the `Compile` class defines a set of APIs using a similar technique. Unfortunately, again there is no mention of attempting to develop the API into a clean one for generalized purposes of self-descriptive JITs for Jalapeño.

There are other technical differences as well; OpenJIT is architected to be a compiler framework, supporting features such as decompilation, various frontend libraries, whereas it is not with Jalapeño. No performance benchmarks have been made public for Jalapeño, whereas we present detailed studies of execution performance validating the effectiveness of reflective JITs, in particular memory profiling technique which directly exploits the 'openness' of OpenJIT. Interestingly enough, Jalapeño is claimed to be only targeting server platforms, and not desktop nor embedded platforms. It would be quite interesting to investigate the memory performance of Jalapeño in the manner we have done, in particular to test whether it makes sense to target smaller platforms or not.

Still, the Jalapeño work is quite impressive, as it has a sophisticated three-level compiler system, and their integrated usage is definitely worth investigating. Moreover, there is a possibility of optimizing the the application together with the runtime system in the VM. This is akin to optimization of reflective systems using the First Futamura projection in object oriented languages, as has been demonstrated by one of the author's older work in [17] and also in [18], but could produce much more practical and interesting results. Such an optimization is more difficult with OpenJIT, although some parts of JVM could be supplanted with Java equivalents, resulting in a hybrid system.

There have been a number of work in practical reflective systems that target Java, such as OpenJava[27], Javassist[5], jContractor[15] , EPP[13], Kava[30], just to name a few. Welch and Stroud present a comprehensive survey of Java reflective systems, discussing differences and tradeoffs of where in the Java's execution process reflection should occur[30].

Although a number of work in the context of open compilers have stressed the possibility of optimization using reflection such as OpenC++[4], our work is the first to propose a system and a framework in the context of a dynamic (JIT) compiler, where run-time information could be exploited. A related work is Welsh's Jaguar system[31], where a JIT compiler is employed to optimize VIA-based communication at runtime in a parallel cluster.

From such a perspective, another related area is dynamic code generation and specialization such as [7, 11, 8]. Their intent is to mostly provide a form of run-time partial evaluation and code specialization based on runtime data and environment. They are typically not structured as a generalized compiler, but have specific libraries to manipulate source structure, and generate code in a "quick" fashion. In this sense they have high commonalities with the OpenJIT frontend system, sans decompilation and being able to handle generalized compilation. It is interesting to investigate whether specialization done with a full-fledged JIT compiler such as OpenJIT would be either be more or less beneficial compared to such specific systems. This not only includes execution times, but also ease of programming for customized compilation. Consel et. al. have investigated a hybrid compile-time and run-time specialization techniques with their Tempo/Harrisa system [29, 22], which are source-level Java specialization system written in C; techniques in their systems could be applicable for OpenJIT with some translator to add annotation info for predicated specializations.

7 Conclusion and Future Work

We have described our research and experience of designing and implementing OpenJIT, an open-ended reflective JIT compiler framework for Java. In particular, we proposed an architecture for a reflective JIT compiler framework on a monolithic VM, and identify the technical challenges as well as the techniques employed, including the minimal set of low-level APIs required that needed to be added to existing JVMs to implement a JIT compiler in Java, contrasting to similar work such as Jalapeno. We performed analysis of the performance characteristics of OpenJIT, both in terms of execution speed and memory consumption, using collaborative instrumentation technique between the JVM and OpenJIT, which allowed us to instrument the JIT performance in real-time, and showed that OpenJIT is quite competitive with existing, commercial JIT systems, and some drawbacks in memory consumption during the bootstrap process could be circumvented without performance loss. We demonstrate a small example of how reflective JITs could be useful class- or application specific customization and optimization by defining a compilet which allowed us to achieve 8-9% performance gain without changing the base-level code.

Numerous future work exists for OpenJIT. We are currently redesigning the backend so that it will be substantially extensible, and better performing. We are also investigating the port of OpenJIT to other systems, including more modern VMs such as Sun's research JVM (formerly EVM). In the due process we are investigating the high-level, generic API for portable interface to VMs. The frontend requires substantial work, including speeding up its various parts as well as adding higher-level programming interfaces. Dynamic loading of not only the compilets, but also the entire OpenJIT system, is also a major goal, for live update and live customization of the OpenJIT. We are also working on several projects using OpenJIT, including a portable DSM system[26], numerical optimizer, and a memory profiler whose early prototype we employed in this work. There are numerous other projects that other people have hinted; we hope to support those projects and keep the development going for the coming years, as open-ended JIT compilers have provided us with more challenges and applications than we had initially foreseen when we started this project two years ago.

References

[1] B. Alpern, D. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, and J. J. Barton. Implementing Jalapeno in Java. In *Proceedings of OOPSLA '99*, pages 314–324, November 1999.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *Proceedings of ICPP '99*, September 1999.

[3] M. Atkinson, L. Daynes, M. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM SIGMOD Record*, 25(4), December 1996.

[4] S. Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA '95*, pages 285–299, 1995.

[5] S. Chiba. Javassist — A Reflection-based Programming Wizard for Java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[6] S. Dieckman and U. Holzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *Proceedings of ECOOP '99*, pages 92–115, 1999.

[7] D. R. Engler and T. A. Proebsting. vcode: a retargetable, extensible, very fast dynamic ocde generation system. In *Proceedings of PLDI '96*, 1996.

[8] N. Fujinami. Automatic and Efficient Run-Time Code Generation Using Object-Oriented Languages. In *Proceedings of ISCOPE '97*, December 1997.

[9] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[10] M. Golm. metaXa and the Futre of Reflection. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[11] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An Evaluation of Staged Run-time Optimization in DyC. In *Proceedings of PLDI '99*, 1999.

[12] U. Holzle. Adaptive optimization for Self: Reconciling High Performance with Exploratory Programming. Technical Report STAN-CS-TR-94-1520, Stanford CSD, 1995.

[13] Y. Ichisugi and Y. Roudier. Extensible Java Preprocessor Kit and Tiny Data-Parallel Java. In *Proceedings of ISCOPE '97*, December 1997.

[14] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: The Story of Squeak - A Usable Small talk Written in Itself. In *Proceedings of OOPSLA '97*, pages 318–326, October 1997.

[15] M. Karaorman, U. Holzle, and J. Bruno. iContractor: A Reflective Java Library to Support Design by Contract. In *Proceedings of Reflection '99*, pages 175–196, July 1999.

[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP '97*, pages 220–242, 1997.

[17] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation. In *Proceedings of OOPSLA '95*, pages 57–64, October 1995.

[18] H. Masuhara and A. Yonezawa. Design and Partial Evaluation of Meta-objects for a Concurrent Reflective Language. In *Proceedings of ECOOP '98*, pages 418–439, July 1998.

[19] S. Matsuoka, H. Ogawa, K. Shimura, Y. Kimura, and K. Hotta. OpenJIT — A Reflective Java JIT Compiler. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[20] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. OpenJIT Frontend System: an implementation of the reflective JIT compiler frontend. *LNCS 1826: Reflection and Software Engineering*, 2000 (to appear).

[21] I. Piumarta and F. Riccardi. Optimizing Direct-threaded Code by Selective Inlining. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pages 291–300, June 1998.

[22] U. P. Schultz, J. L. Lawall, C. Consel, and G. Muller. Towards Automatic Specialization of Java Programs. In *Proceedings of ECOOP '99*, June 1999.

[23] K. Shimura. OpenJIT Backend Compiler. http://www.openjit.org/docs/backend-compiler/openjit-shimura-doc-1.pdf, June 1998.

[24] K. Shimura and Y. Kimura. Experimental development of java jit compiler. In *IPSJ SIG Notes 96-ARC-120*, pages 37–42, October 1996.

[25] K. Shimura and S. Matsuoka. OpenJIT Backend Compiler (Runtime) Internal Specification version 1.1.7. http://www.openjit.org/docs/backend-internal/index.html, October 1999.

[26] Y. Sohda, H. Ogawa, and S. Matsuoka. OMPC++ — A Portable High-Performance Implementation of DSM using OpenC++ Reflection. In *Proceedings of Reflection '99*, pages 215–234, July 1999.

[27] M. Tatsubori and S. Chiba. Programming Support of Design Patterns with Compile-time Reflection. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[28] Stanford University. SUIF Homepage. http://www-suif.stanford.edu/.

[29] E. N. Volanschi, C. Consel, and C. Cowan. Declarative Specialization of Object-Oriented Programs. In *Proceedings of OOPSLA '97*, pages 286–300, October 1997.

[30] I. Welch and R. Stroud. From Dalang to Kava - the Evolution of a Reflective Java Extention. In *Proceedings of Reflection '99*, pages 2–21, July 1999.

[31] M. Welsh and D. Culler. Jaguar: Enabling Efficient Communication and I/O from Java. *Concurrency: Practice and Experience*, December 1999. Special Issue on Java for High-Performance Applications.

Using Objects for Next Generation Communication Services

Munir Cochinwala

Telcordia Technologies, 445 South Street, Morristown, NJ 07960 USA
munir@research.telcordia.com

Abstract. The integration of the telephone network and the internet enables convergence of voice and data services. The explosion of information appliances also provides new service opportunities. Object-oriented systems show great promise for building new classes of services that dynamically adapt to changing networks and appliances. This paper describes how we use migratory objects to build new classes of services. The focus is on telecommunication services where voice is integrated with other types of media/data.

1 Introduction

Recent dramatic improvement in data networking technologies have dramatically changed the landscape for traditional providers of telecommunication services. Data networks are now able to provide adequate quality telephone service, both in access networks (e.g. IP over cable) and in backbone networks (e.g. various forms of IP telephony over ATM). Telecommunication providers see a significant threat to their revenue. The presence of IP telephony providers and open competition in the long distance market has resulted in price reductions ranging from from 20% in the U.S. to 70% in Germany.

However, significant new revenue opportunities are available to telecom providers who can expand into new markets. The market size is significant. Consumer spending on communication services such as telephony, cable, internet access and premium services such as Video on Demand are predicted to be around $100 billion in 2001 with telephony services accounting for $48 billion.

The underlying technological trend is the explosive growth of data networks. The percentage of network bandwidth occupied by voice traffic is predicted to decline from 40% today to less than 5% in 2005. The optimal strategy for telecommunication may be handle voice as much as possible like data.

This new integrated network is termed the Next Generation Network (NGN). A NGN is a network where all kinds of information (voice, fax, video, data) are transmitted uniformly using packet-based transport and switching media.

The challenge for telecom providers is to deploy new, innovative services that integrate the traditional ease of use and reliability of the phone network with the vast new potential of the data network and the web.

Using the flexibility of NGN, a new integrated multi-service communications framework can be created. Using this framework telecommunication companies

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 388–393, 2000.
© Springer-Verlag Berlin Heidelberg 2000

can bundle new services with traditional services and reverse the trend of losing revenues to internet, cable and entertainment companies.

At Telcordia we have developed a framework for NGN services based on objects and object migration. Our goal is to develop a framework where new services can be rapidly developed and deployed. We believe that our framework meets goals of seamless integration of the phone network and data network incorporating the best that each network has to offer. The new framework requires a significant change in the infrastructure used for call (communication) set-up.

2 Object Framework and Applications

Our goal is to provide a framework where new services can be developed and deployed rapidly. Our focus is more than simple telephone calls, it is communicating applications. New application should be able to use the new services as easily as they would make a telephone call.

We have designed a basic set of building blocks that can be assembled to create new services. A communicating service begins with a pair of coupled objects which are the communication endpoints. For a typical 2-party communication, one endpoint is sent to the destination party. A third party call set-up would require sending both sides to different destinations. Local resource managers are responsible for allocation of resources such as audio devices for normal telephone calls or screen real estate, camera, etc for more complex interactions. The communicating service can also be created by cloning an existing communicating service.

To place an ordinary telephone call, a user creates two endpoints, sends one to the intended recipient, attaches the other to a local device (e.g., a phone). When the peer arrives on the recipient's desktop, it will attempt to get resources and give an alert (e.g., ring the phone). If the originating user does not immediately attach his endpoint to a device, we are left with a symmetric situation resembling a "hot line": each side has an endpoint for the call, but no resources (or signaling) will take place until one side picks up. This can be used to implement a variety of interesting services (e.g., an anonymous dating service).

The endpoint objects can be migrated to different devices or destinations. We use object mobility as the basis for migration. One might ask why we bother using mobility at all, rather than just using standard distributed object technology (e.g., CORBA). From our perspective, object mobility is simply distributed objects, along with (1) transparent message forwarding, and (2) automatic code distribution. The second capability is not necessarily critical in our system (where we emphasize object composition over new code creation). For our purpose, the main advantage of automatic message forwarding is that it enables greater decoupling between clients and servers.

A user may wish wish to integrate a number of service capabilities into a single handler. For example, he might have a number of objects able to display media of different types, and one of these may be chosen for an incoming media stream. Or he might have access to a number of different service providers offering a common capability (e.g., bit transport), and he wants to choose the one offering the best rate. Such management policies can be embedded in specialized objects, which we call ”service containers”. These containers behave like ordinary desktop containers (i.e., managed objects can be dragged in and out), and provide coherent presentation and management of their contained services.

We have focused on building framework objects that can be used to implement complex business function. For example, object containment can be used to implement complex services such as bundling voice and data. Object containment can also be used to maintain application context when items are passed among participants. We have used these concepts to implement a tele-medicine application.

Using our framework, the tele-medicine application can be designed so that even non-technical people like doctors and nurses can make effective use of the flexibility of services. The doctor can make a phone call to another doctor, patient or pharmacist. During the conversation, any one of the participants can choose to share or include a data item and use the phone call as a channel to share or broadcast data to the other participants. Private communication between any subset of the parties can also be easily accommodated by establishing a private channel or by using private data in objects. All of these functions can be accomplished within the phone call or any other communication. The phone call is just another application and to the end-user or the developer, it is a convenient channel that is currently configured to handle audio. The channel can dynamically adapt to other media types.

In our prototype, we demonstrate the following features:

- Information passed between patient, doctor and pharmacist is encapsulated in objects
- Prescription and patient history are bundled with voice communication and migrated across the parties
- Database access and transaction context of patient history can be shared and protected
- The infrastructure supports customized handling of media types based on user preferences. These preferences may be to send data to one network while the voice goes to another network.

3 Services Model

We now compare our object-based framework to traditional implementation of telephony services. We focus on the differences in call set-up and call models.

Current telephony systems are based on centralized creation and deployment of relatively static services, carefully designed to interoperate safely. Conversely, on the Internet, anybody can create and deploy a new service, but these services cannot interoperate effectively.

In traditional telephony, there is a sharp division between services and sessions. Services are usually written in a programming language, carefully integrated with other system components, with severe security requirements, while a session is created by a user (or a service), and has a relatively short duration/value. An important thrust of our work is to eliminate the distinction between these two types of entities.

In telephony, it is common to use very different protocols for the originating and terminating endpoints of a call. This approach greatly complicates the design of many services (e.g., third-party call setup). Instead, we try to make use of call models that are completely symmetric.

We have used object and object migration as the fundamental method of communication. We now present a comparison of the our approach compared with the traditional approach for telephony.

- In traditional telephony call setup is based on a universal addressing scheme. That is, the initiator decides on the address of the responder (possibly after some universal number translation phase, like 800 number translation, or some multilevel lookup, as in call forwarding or internet DNS), and requests a connection to that endpoint.

  In our framework, all application addressing is to objects; the client never sees where the message is actually routed. Objects can migrate while the system is running; this migration is transparent to clients/communication peers.
- Traditionally, there is a previously agreed-upon protocol regulating communication between the initiator and responder; this protocol regulates how parties can join, suspend, and terminate a call, what sort of busy/failure signals can be conveyed, etc. This makes changes to the call model extremely difficult; evolutions to the call model are very rare, and must always maintain backward compatibility.

  In our system, the call initiator chooses the complete protocol to be used for the communication. Of course, by choosing a protocol where the terminating endpoint is expected to provide a standard terminating half call model, he can get the effect of the current call model. However, he can also choose a completely different call model, requesting direct interaction with devices at the terminating endpoint.
- Telephone calls require initiating and terminating sides to have very different roles. This is because conventional calls are relatively short lived, during which the initiator (but not necessarily the terminator) is assumed to be participating.

  Conversely, our call models are generally symmetric. This is because the calls are possibly long-lived, with either or both parties temporarily or permanently suspending participation, without the call being torn down (from the logical standpoint - an implementation may choose to tear down the actual circuit for efficiency reasons).
- Audio is the media of choice in telephony and is normally determined at call setup time (perhaps through a process of negotiation between the two parties). In contrast, in our calls, the initiator (or, more commonly, the call model) obtains a handle to a virtual environment of devices (resources)

  belonging to the terminator. The call model can allocate and/or deallocate resources, on either side of the call, at any time.

  Resource allocation in our framework is carried out through resource managers. A resource manager is an object that can be asked to deliver other objects, given a description of the object desired (typically its type). When a call (or any other communication object, such as a document or an advertisement) wants to make use of resources (such as stable storage, a speaker, or an alerting device like a ringer), it gets suitable objects from the resource manager. (Because devices like phones can themselves be viewed as being composed of many subdevices, the resources are themselves resource managers.)

- Security is maintained by associating actions with principals (typically representing people or organizations), and annotating sensitive resources (like files and devices) with lists of principals (called access control lists). When an action tries to access a resource, the system checks that the principal responsible for the action is on the list of allowed accessors; if the principal is not on the list, the access fails. However, this makes it difficult for one party to hand access off to another, which is an essential paradigm in our system.

  Our system uses what is called capability-based access control: a handle to an object itself guarantees access. Since handles can be freely copied and sent to other principals, this allows one participant to hand access off to another participant.

  The usual problem with capability-based access control is what is called the confinement problem: once you give away a capability, you can never get it back. We solve this problem by making extensive use of a proxy discipline. When a resource manager wants to delegate use of a resource to some party that it does not fully trust, it creates another object of the same type that forwards requests on to the original resource. However, this new resource also has a ”kill” switch, accessible only to the resource manager, that can be used to dissociate the proxy object from the original. (The proxy is also designed so that when it gives out a subresource, these subresources are also given out as proxies, which can be killed when the original proxy is killed.) This allows the resource manager to revoke previously issued resource grants at the cost of a level of indirection). (It can also preprogram the proxies to kill themselves, either after some time interval or after a certain number of accesses.)

4 Conclusion

We have a prototype built using the framework. We have incorporated multiple devices in our prototype ranging from Java Rings, MP3 players, IP phones, and gateways to traditional telephone networks. The infrastructure allows setting up communication channels, attaching resources (e.g audio) to communication channels and migration of live communication across devices and users. Furthermore, a complex object (voice and data) can be combined or split and sent to different destination (users) on different networks. The immediate next steps are to incorporate our framework into NGN products and deploy the products.

Deployment of the framework and applications on a world-wide basis poses tremendous challenges. Challenges from traditional system design are the scalability and reliability of the new applications. Security is a major concern that we have left unexplored. Definition of a security policy and implementation of security for objects that can appear in a user's environment are avenues for exploration.

We have object mobility as the basis for communication. The implementation and definition of the object mobility platform is an area of further research. A simple question is how much of an object to move when the top level object moves. Moving the complete object may have performance impacts while moving a partial object leaves room for failure due to network partitions or object repository failure.

Finally, telecom and network providers have to design schemes for managing applications that are long-lived and may require high bandwidth. Monitoring and managing these applications may require cooperation between application objects and network management software for efficient allocation of resources.

Empirical Study of Object-Layout Strategies and Optimization Techniques

Natalie Eckel* and Joseph (Yossi) Gil**

Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel
natalie | yogi@cs.Technion.AC.IL

Abstract. Although there is a large body of research on the time overhead of object oriented programs, there is little work on memory overhead. This paper takes an empirical approach to the study of this overhead, which turns out to be significant in the presence of multiple inheritance. We study the performance, in terms of overhead to object size of three compilation strategies: separate compilation, whole program analysis, and user annotations as done in C++. A variant to each such strategy is the inclusion of pointers to indirect virtual bases in objects. Using a database of several large multiple inheritance hierarchies, spanning 7000 classes, several application domains and different programming languages we find that in all strategies there are certain classes which give rise a large number of compiler generated fields in their object layout. We then study the efficacy of the recently introduced inlining and bidirectional object layout optimization techniques, and show that an average saving of close to 50% in this overhead can be achieved.

1 Introduction

One of the most difficult challenges in the implementation of object oriented programming languages is to efficiently realize together multiple inheritance (MI) and dynamic dispatch. Indeed, it was believed [8] that it was impossible to efficiently introduce MI into C++, until proved otherwise [26].

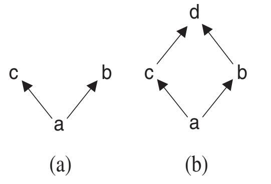


Fig. 1: Multiple inheritance and virtual inheritance.

The main difficulty is due to the fact that in presence of MI, an object of class a, can also serve as an instance of classes b and c, where no inheritance relationship exists between b and c, as in Fig. 1(a). The problem is complicated further in case b and c have a common ancestor, d, as depicted in Fig. 1(b). In this case d is called a *virtual base*, (of b, c, and a), and the inheritance links between b and d and c and d are called virtual inheritance[1] (VI).

To support this highly polymorphic nature of objects in presence of MI, the compiler is required to generate extra data fields in objects. Benchmarking shows that this overhead is significant [28]; and it can even double the memory footprint of some applications.

* Contact author

** Work done in part during a visit to the IBM T.J. Watson Research Center

[1] also *shared inheritance*

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 394–421, 2000.
© Springer-Verlag Berlin Heidelberg 2000

In a recent theoretical paper, Gil and Sweeney [11] described the kinds of compiler generated fields: VPTR*s* —pointers to virtual functions tables (VTBLS), and VBPTRS—pointers to virtual bases. There are two kinds of VBPTRS. *Essential* VBPTR*s* , or for short, *e*-VBPTRS, point to *direct* virtual bases. A time-space tradeoff is offered by *inessential* VBPTR*s* , *i*-VBPTRS for short, which point to *indirect* virtual bases, i.e., virtual bases of virtual bases. *i*-VBPTRS make it possible to access an indirect virtual base in a single dereference operation, without going through any intermediate virtual bases. Optimization techniques for reducing the number of these fields were offered in the theoretical framework of [11].

Taking an empirical, yet language-independent, approach, our work continues the study of compiler generated fields. Compiler generated fields can be thought of as the *per-object* memory overhead of MI. It should be noted that MI incurs a bloat in two other kinds of memory overheads: *per-class* for storing class tables, and *per-hierarchy* for storing e.g., type inclusion information. Both are left outside the scope of this paper: Per-class overhead, which tends to be small, was thoroughly studied in the context of C++ in [28]; optimization techniques were suggested e.g., in [9]. See [15] for an empirical and theoretical study as well as a survey of research of the per-hierarchy overhead.

As described in Sec. 2, our study relies on a database of circa 7,000 classes. The main issues which interest us are summarized in the next few paragraphs. These also give forward references to the detailed discussion in the body of the paper.

Topology of large inheritance hierarchies. With such a large number of classes, it is becoming difficult to have an intuition on the structure of hierarchies. In Sec. 3 we define a suite of parameters of the topology of the directed acyclic graph (DAG) of inheritance. These parameters should help in gaining better understanding of the structure of inheritance hierarchies. By applying this suite to our database, we are able to reach some interesting findings on the topology of inheritance hierarchies, and explain some of the benchmarking results.

Separate compilation vs. whole program analysis. VI complicates MI not only by its presence, but also due to the uncertainty it generates. A compiler encountering class y inheriting from x, cannot be sure that x is not a virtual base of y before verifying in the whole inheritance hierarchy that there is no other class z that inherits *independently* from *both* x and y. In a separate compilation environment, the compiler is therefore inclined to make a worst case assumption and treat *all* inheritance links as virtual.

This research is the first to investigate the cost of this assumption which is a direct consequence of the separate compilation model. We will see in Sec. 4.4 that separate compilation more than doubles the number of compiler generated fields of a median object, while for some objects it incurs more than 40-fold increase in the number of the compiler generated fields!

In C++, the responsibility of distinguishing between virtual and non-virtual links lies with the programmer. However, this language design decision, which is probably as questionable as many other features of the language [24], does not make the situation much better. As explained in [11], with the absence of omniscient capacities, the programmer will tend to conservatively choose virtual over non-virtual inheritance. For example, in the almost standard IDL hierarchy (used e.g., in [10, 4, 28]), out of the 65 inheritance links (connecting 66 classes), 50 links were marked as virtual. Evidently, a whole program analysis reveals that only two inheritance links in this hierarchy are truly virtual!

Inessential VBPTR*s.* As explained in [11], all VI edges must be represented in the *traditional object layout scheme* using an *e*-VBPTR. However, to avoid an excessive number of dereference operations, which are known to be expensive in modern RISC processors, many compilers place *inessential* VBPTRS in object layouts. In Sec. 4.3 we give estimates on the actual memory overhead of this compiler implementation decision. We also comment on the expected number of dereference operation required to reach a virtual base in the absence of *i*-VBPTR.

Optimization techniques. Our findings show that even in the frugal whole program analysis strategy, there is a significant fraction of mammoth classes—classes with around a hundred compiler generated fields in each of their instances. This number increases to circa 250 if *i*-VBPTRS are included!

It was shown in [11] using a theoretical analysis and a number of small case studies that a significant reduction in these numbers might be possible, using a number of optimization techniques.

One of our primary objectives is to understand these optimization techniques better and to apply them to show that the memory overhead induced by the time-efficient *i*-VBPTRS can be reduced to a minimum. To this end, we give an empirical study of the efficacy of the following object layout optimization techniques.

ETRANS *Elimination of transitive virtual inheritance edges.* This technique is used mainly as a pre-processing stage for other, more advanced techniques.

DEVIRT *Devirtualizing single virtual inheritance edges.* Single virtual inheritance edges are edges designated as virtual by the programmer, but not serving as such.

S-INLN *Simple inlining of virtual bases. Inlining* means that no VBPTR is used to represent a virtual base in its descendant. Instead, the corresponding subobject is made part of the including object. The inlining is *simple* if no attempt is made to inline a virtual base into more than one of its descendants.
Note that devirtualization can also be thought of as inlining.

A-INLN *Aggressive inlining of virtual bases.* In aggressive inlining, a *maximal independent set* algorithm is used to maximize the number of descendants into which a virtual base is inlined. Since the best such algorithms are still exponential, care should be taken before their application. We therefore dedicate some attention in Sec. 6.2 to estimate the resources required for applying this advanced technique.

PAIRUP *Marriage of immediate non-virtual bases.* The bidirectional object layout suggested in [11] makes it possible to “marry” together a positive and an negative immediate, non-virtual bases of a class, whereby saving one VPTR in their representation. The algorithm for doing so is described in procedure *Pair-Up* of [11].

EPHEM *Ephemeral marriage of virtual bases.* Yet another optimization technique which uses the bidirectional object layout is the ephemeral marriage of virtual bases. The marriage is called *ephemeral* since virtual bases married in one class are not necessarily married in any of it descendants.

BIDIR *Combination of* PAIRUP *and* EPHEM.

H-PAIRUP *Marriage of immediate, non-virtual, hermaphrodite bases.*

This paper also proposes and investigates yet another new technique for the optimization of object layout. The *hermaphroditing* idea is similar to bidirectional object layout, except that directed classes, may have instances of both positive and negative directionality. Hermaphroditing increases further the marriage opportunities, but also requires means to distinguish at runtime between negatively and positively oriented objects. For example, this distinction could be made by using the sign bit of object pointers, so that negative pointers would be used to designate objects laid out in the negative direction. Modifications to the dispatch mechanism would be required to make the distinction between negatively and positively oriented VTBLS. H-PAIRUP refers to the hermaphrodite version of PAIRUP.

H-EPHEM *Ephemeral marriage of hermaphrodite virtual bases.*

This is the hermaphrodite version of EPHEM.

H-BIDIR *Combination of* H-PAIRUP *and* H-EPHEM.

In other words, this is the hermaphrodite version of BIDIR.

Table 1 gives the summary of the requirements and the potential savings of all of these techniques.

Technique	Requirements		Savings	
	whole program analysis	architecture support	VPTRS	VBPTRS
ETRANS	-	-	+	+
DEVIRT	+	-	+	+
S-INLN	+	-	+	+
A-INLN	+	-	+	+
PAIRUP	-	-	+	-
EPHEM	-	-	+	-
BIDIR	-	-	+	-
H-PAIRUP	-	+	+	+
H-EPHEM	-	+	+	+
H-BIDIR	-	+	+	+

Table 1: Techniques for optimizing object layout.

We see that inlining of all variants: devirtualization, simple and aggressive, requires whole program analysis. The primary kind of savings of savings is in VBPTRS. However, each saving in a VBPTR bears a potential for VPTR saving due to sharing. The efficacy of the inlining family of optimization techniques is discussed in Sec. 6.

On the other hand, bidirectional object layout techniques, whose benchmarking is described in Sec. 7, do not require whole program analysis, and can only save VPTRS. It is a rather surprising conclusion of our study that the savings of bidirectional techniques are in the same order of magnitude as DEVIRT and as A-INLN. Hermaphroditing is a new optimization technique described in detail in the sequel. The reader is referred to [11] for a detailed description of the other techniques. The efficacy of all the optimization techniques is summarized in Sec. 8. Finally, Sec. 9 gives the conclusions and outlines directions for further research.

2 Experimental Setting

This section discusses the database used in our benchmarking. Prior to this description, few words are in place regarding the special problems in making measurements of the sort we are interested in. The main difficulty in benchmarking memory overhead due to MI is that of scale. Although measurements of time made on small and medium sized benchmark programs such as SPEC [12] and SPECjvm98 [25] remain meaningful when extrapolated to large application, no similar extrapolation can be made for memory usage. Small programs tend to make little use of MI. Consequently, their typical object layout is small and simple, leaving little or no room for optimization. Consider for example Driesen and Hölzle timing of virtual functions calls [10]. Their second largest application, groff, has only 92 classes, with only 48 inheritance relationships, with no MI, let alone VI. This data-point of 19,000 LOC of C++ is useless for our purposes, since in the absence of MI, all three kinds of memory overhead are minimal. In fact, the per-object memory overhead is reduced to one compiler generated field [11].

Obtaining a sample of large applications that make use of complex inheritance hierarchies, is difficult since such applications tend to be *propriety*, prohibitively *expensive*, and *unavailable* at all in source code form. Yet another bias in sampling is that as a result of the significant overhead of MI as encountered by every user, enforced by warnings from industry leaders (e.g., [5, 17, 18]), designers tend to restrain their needs for MI. Reducing this overhead by work such as this one is likely to increase the use of MI.

Our database selection started from the collection of large hierarchies which served Krall, Vitek and Horspool in benchmarking their algorithms for efficient type inclusions tests [16, 15].[2] This collection was used in other classical benchmarking papers, e.g., [9]. Of the eleven applications used in [15], four did not make any use of MI, and hence are not suitable for object-layout benchmarking (nor are they particularly appropriate for demonstrating type inclusion algorithms).

Also, a careful examination of the 225 nodes in the Java hierarchy used in [16, 15], shows that it wrongly assumes that Java interfaces inherit from class Object. We therefore disregarded this hierarchy, and replaced it with a hierarchy of around 1,700 classes of JDK 1.1. Since Java uses only a very restricted form of MI, we did not use this Java hierarchy at all in comparing the different strategies of MI. We however used it in our study of the topological structure of MI.

To these seven hierarchies, we added the class hierarchy of ISE version 4 [14] distribution of Eiffel [20]. Overall, the database consists of real word applications, covering a spectrum of usage of MI in various programming languages: C++ [27], Java [3], Eiffel [20], LOV (a language similar to Eiffel due to Verilog, France, a manufacturer of an OO CASE tool), Self [29] as well as Laure language of Caseau [6].

The hierarchies used are summarized in Table 2. The most important characteristics of each hierarchy are n, the number of classes (nodes), and m the number of inheritance relationships (edges). In total, the database has circa 8,500 nodes and 11,500 edges. Of these, around 7,000 classes and 9,500 edges were used in benchmarking MI.

Column r in the table gives the number of roots, i.e., classes which do not inherit from any other class. In single inheritance hierarchies r is simply the number of connected components, which will be 1 in the case the hierarchy is a tree. As can be seen from the table, r is strongly dependent in the programming language. Languages which allow multiple roots have a fair number of roots. However, overall only 2.9% of all classes are roots.

[2] Our thanks to Jan Vitek for providing access to this data.

The next column, α, the average number of parents of non-root classes, is indicative of the extent to which MI is used.

Hierarchy	Language	$n^{(i)}$	$m^{(ii)}$	$r^{(iii)}$	$\alpha^{(iv)}$
Unidraw	C++	613	476	147	1.02
Self	Self	1801	1838	51	1.05
Laure	Laure	295	315	1	1.07
JDK1.1	Java	1654	1927	89	1.23
Eiffel4	Eiffel	1999	2678	1	1.34
Ed	LOV	434	750	1	1.73
LOV	LOV	436	774	1	1.78
Geode	LOV	1318	2785	1	2.11
Total		8550	11543	293	1.39

(i) Number of classes
(ii) Number of inheritance edges
(iii) Number of root classes
(iv) Average number of parents of non-root classes

Table 2: Class hierarchies used in our experiments.

For all single inheritance hierarchies, $\alpha \equiv 1$. The hierarchies in Table 2 are sorted in ascending order of α. Thus, Laure, Self and Unidraw hierarchies are pretty close to a single inheritance hierarchy, while all the three applications written in LOV make an extensive use of MI.

It should be announced that some of the numbers presented in Table 2 *do not* agree with the corresponding values in Table 2 in [16] and Table 2 in [15]. The value of n for Unidraw, Self and Geode in Table 2 is greater by one than in [16, 15]. Also, the average number of parents (α) for Unidraw, Ed, LOV and Geode, does not agree with [16, 15], with differences ranging from 0.01 to 0.22. In view of these differences, we verified our software by having it rewritten by another person in another programming language; the results turned out to be the same in both implementations. This discrepancy could be explained by updates made by the authors to their database subsequent to publication, rather than a flaw in the experimental work of [16, 15]. However, it should be noted that the numbers in our table do agree with earlier reports on the same data (Table 1 of [9]).

Our data did not include information on the use of virtual vs. ordinary (repeated) inheritance in C++, nor on the equivalent of this distinction in the Eiffel family of languages. We were therefore inclined to assume that no repeated inheritance occured, i.e., no class is duplicated in any of its descendants. Thus, in the hierarchy of Fig. 1(b), we *always* use the semantics that there is only one occurrence of a subobject d in a, even though C++ gives way also to the semantics of two such occurrences. Our assumption agrees with the observation that repeated inheritance is a rarity, with the belief of some that “repeated inheritance is an abomination" [3], and the fact that many programming languages do not even support it, or as it is the case with Eiffel, require the programmer to go into extraordinary effort to make use of it.

No statistics on runtime memory usage was available. Indeed, for a library hierarchy such data would be meaningless, unless collected over a very broad array of applications which make use of this library. Moreover, measurement of memory usage in runtime opens the door to a debate between conflicting definitions, such as high water mark and total memory usage. Even further, many subtle issues such as fragmentation, tradeoffs between sizes of classes which tend to be instantiated together, etc., have to be dealt when dynamic runtime information is taken into account. Consequently, this research conten-

[3] words of an anonymous reviewer to [11]

ded itself with measurements based on static analysis; the problem of benchmarking runtime information is left for further research.

Our results are presented broken by overhead to object size and sometimes by hierarchy. The fundamental hypothesis is that since frequency of instantiation each class cannot be predicted, all the layout of *all* classes should be optimized. To obtain a single number summarizing a range of measurements it is sometimes convenient to make a

> UNIFORM INSTANTIATION HYPOTHESIS: *All classes are equally likely to be instantiated.*

In other words, with the absence of any other information, there is no reason to assume that one class is more likely to be instantiated than another class. Another hypothesis which can be used to compute such a single number is:

> LEAF INSTANTIATION HYPOTHESIS: *All leaf classes are equally likely to be instantiated, but internal classes are never instantiated.*

This hypothesis is supported by the observation that an abstract class is (usually) not a leaf, and on the proviso that library-user defined classes which are likely to be instantiated more frequently, tend also be leaves. As we will see, the differences in numbers between these two hypothesis is minute. This strengthens our belief that the static results we obtain carry their meaning to runtime performance.

3 The Topology of MI Hierarchies

Hierarchy	μ[i]	ℓ[ii]	b[iii]	d[iv]	v[v]	ν[vi]
Unidraw	7.2%	78.5%	9	8	2	0.3%
Self	21.1%	63.0%	10	16	3	0.2%
Laure	3.5%	64.1%	8	11	11	3.7%
JDK1.1	19.3%	77.3%	10	8	19	1.1%
Eiffel4	23.4%	61.6%	10	14	63	3.2%
Ed	5.1%	50.7%	8	8	23	5.3%
LOV	5.1%	50.0%	8	9	24	5.5%
Geode	15.4%	55.5%	10	11	100	7.6%
Total	100%	64.1%	13	16	245	2.9%

[i] The relative weight of this hierarchy in the database $n/\sum n$

[ii] Percentage of leaf classes

[iii] Depth of the complete binary tree with n nodes

[iv] Depth of the hierarchy

[v] Number of virtual bases

[vi] Percentage of virtual bases

Table 3: Topological properties of the class hierarchies.

In designing algorithms and heuristics for MI hierarchies, it is important to understand the patterns of usage of MI in real applications.

Since such hierarchies are huge, it is difficult to gain any insight into their structure by a drawing their DAG which is often highly non-planar. Instead, we propose to study the structure of MI hierarchies by defining and computing topological properties of the inheritance DAG. Some such properties are summarized in Table 3.

Column μ gives the weight of this hierarchy in the database, computed based on the number of classes. Column ℓ gives the percentage of leaf classes. We already saw that there are very few root classes; now we see that the majority of all classes are leafs. This is a strong indication that hierarchies do not have a lattice like structure, despite the counter argument based on proper design considerations [13]. Perhaps this is also the reason why Caseau [7] algorithm for type inclusion tests which tried to impose a lattice structure on the graph was inferior to other attempts at this challenge.

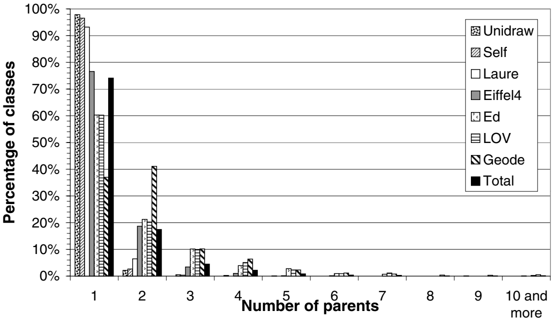


Fig. 2: Distribution of number of immediate base classes of non-root classes.

The depth of each hierarchy, i.e., the maximal distance in edges between a root and a leaf is denoted by d. By comparing d with the depth of a complete binary tree ($\lceil \lg(n+1) \rceil - 1$) we see that the hierarchies are very shallow. In fact, the depth never exceeds that of a balanced binary tree such as AVL.

The total number of virtual bases, i.e., classes which serve as an ancestor to some other class along more than one path, is given in column v of Table 3. Note that the number of virtual bases is relatively small. Overall, only about 3% of all classes are virtual bases. However, as we shall see later, the majority of classes have at least one virtual base. As a result, this small fraction of virtual classes has a huge impact the object size of many more classes.

Some more topological properties of our hierarchies are given in Table 4.

Hierarchy	α(i)	β(ii)	γ(iii)
Unidraw	$1.02^{\pm 0.15}$	$3.61^{\pm 5.18}$	$0.02^{\pm 0.18}$
Self	$1.05^{\pm 0.33}$	$2.76^{\pm 11.2}$	$0.73^{\pm 0.47}$
Laure	$1.07^{\pm 0.27}$	$2.97^{\pm 1.77}$	$2.86^{\pm 1.15}$
JDK1.1	$1.23^{\pm 0.57}$	$5.13^{\pm 22.9}$	$0.52^{\pm 0.74}$
Eiffel4	$1.34^{\pm 0.75}$	$3.49^{\pm 18.8}$	$2.49^{\pm 2.61}$
Ed	$1.73^{\pm 1.17}$	$3.50^{\pm 7.10}$	$3.79^{\pm 2.79}$
LOV	$1.78^{\pm 1.28}$	$3.55^{\pm 7.22}$	$3.99^{\pm 3.04}$
Geode	$2.11^{\pm 1.50}$	$4.76^{\pm 17.1}$	$8.37^{\pm 6.30}$
Total	$1.39^{\pm 0.92}$	$3.74^{\pm 15.6}$	$2.62^{\pm 4.02}$

(i) Average number and standard deviation of parents of non-root classes

(ii) Average number and standard deviation of children of non-leaf classes

(iii) Average number and standard deviation of virtual base classes

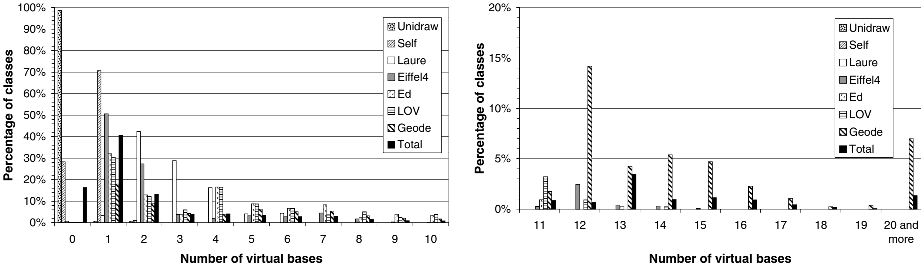
Table 4: Additional topological properties of the class hierarchies.

Column, α is similar to the column with the same title in Table 2, gives both the average and the standard deviation of the number of parents of non root classes. Note that both these values are not much greater than 1. By examining the distribution of the number of parents in Fig. 2, we see that more than 70% of all classes have only one parent, while for the Unidraw, Self and Laure hierarchies, this number increases to over 90%. Further, more than 90% of all classes have no more than two parents. Even for Geode, which has the least number of classes with one parent, close to 80% of all classes have at most two parents. We see that classes with large number of parents are rare, even in hierarchies which make quite an extensive use of MI.

We argue that the impact of MI on classes is gradual. Although classes tend to have a small number of immediate parents, these parents might also use MI, etc. This phenomena can be seen by examining the number of root-classes from which non-root classes inherit, directly or indirectly, which is $20.59^{\pm 4.49}$ in Geode, even though for this hierarchy $\alpha = 1.05^{\pm 0.33}$. Interestingly, for this hierarchy, $\alpha^d = 1.05^{16} = 2.1827$, which is still much smaller than the average number of roots. In other words, the nodes in which multiple inheritance occurs are located in critical and influential junctions of the hierarchy.

Dual to the α parameter is β, the number of children of non-leaf classes. As can be seen from Table 4, the standard deviation of β is quite large, often greater than its average value.

Finally, the γ column of Table 4 gives the average number of virtual bases, direct and indirect that each class has. The distribution of the number is depicted in Fig. 3. In Unidraw there are very few classes with virtual bases. This is partly explained by the fact that as many as 24% of the classes of Unidraw are roots. With this exception, we see that even though virtual bases are scarce in general, many more classes have virtual bases. In the Self hierarchy for example, even though there are in total only 3 virtual bases, over 70% of all classes use one or more of these virtual bases.



(a) 1. . . 10 virtual bases

(b) 11. . . 20 and more virtual bases

Fig. 3: Distribution of number of virtual bases of a class.

In single-root hierarchies, virtually all classes have at least one virtual base. Further, in Eiffel4, 27% of all classes have two virtual bases. There is also a significant fraction of classes with a very large number of virtual bases. In Geode over 40% of classes have 10 or more virtual bases. The large standard deviation in the number of virtual bases is also worth noting, since the overhead in object size may increase quadratically with the number of virtual bases [11].

4 Traditional Compilation Strategies

There are two major strategic decisions in the compilation of MI. First, it should be decided whether whole program analysis could be done, at the cost of slowing down the compilation process. We will use the notation WHO for a whole program analysis compilation strategy.

The alternative to WHO is to use a separate compilation strategy, which will be denoted as SEP. SEP which appears to be used by compilers such as ISE Eiffel [19, pp.339-340], is faster, but may produce less efficient executable. In particular, since a

SEP compiler has no way of predicting that an inheritance edge will not turn out to be virtual, it must conservatively assume that *all* inheritance edges are virtual.

The traditional C++ compilation model, denoted as CC, sits somewhere in between SEP and WHO. On the one hand, the programmer must annotate all virtual inheritance as such by using the virtual keyword. On the other hand, more advance optimization techniques are impossible since no whole program information is available until link time.

This section focuses on CC and SEP. As we shall see shortly, WHO is nothing but CC with the application of devirtualization. We postpone the discussion of WHO to Sec. 6.3.

The second decision that must be made is whether *i*-VBPTRS be included in object layout. The advantage is in maintaining the time efficient "single dereference distance" between an object and each of its subobjects. A "+" superscript will be used to denote a strategy variant which uses *i*-VBPTRS: SEP$^+$, WHO$^+$, and CC$^+$.

The alternative is to optimize memory by omitting those, but with time overhead of following a chain of pointers in an upcast to a non-direct virtual base. A "−" superscript on the name of strategy will be used to denote omission of *i*-VBPTRS.

4.1 Kinds of Inheritance Edges

To simulate the CC in our hierarchies which are mostly non-C++, it is necessary to *automatically* generate the programmer's virtual annotations.

In general, it is difficult to predict this kind of human decisions, which might be whimsical or influenced by a personal sense of style. For example, the IDL hierarchy unnecessarily used a large number of virtual inheritance, since it is designed also to be an elegant application framework.

In our to understand how programmer annotations are generated automatically consider the hierarchy graph of Fig. 4. It makes sense to assume that edge (b,a) will *not* be annotated as virtual. We will automatically denote edges of this sort as *non-virtual*. We have that (f,d) and (f,e) are non-virtual as well. With our assumption forbidding repeated inheritance, we have that (d,b), (e,b) and (f,b) are virtual. Edge (c,b) poses an interesting dilemma. A whole program analysis would reveal that this edge is non-virtual. However, a prudent programmer is likely to mark it as virtual in anticipation of code evolution. A class which served once as a virtual base is likely to serve as such again in the future. This is confirmed by our findings in Sec. 3 that virtual bases tend to serve multiple times as such. In this example, the introduction of a new class g inheriting from both c and d will *require* that (c,b) is marked as virtual. We will call (c,b) and other edges of its kind *potentially virtual*. In order to simulate the CC strategy we assume that potentially virtual edges are annotated virtual. In WHO, these edges will be considered non-virtual.

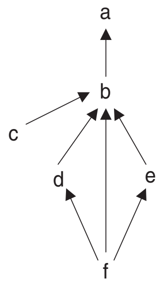


Fig. 4: The three kinds of inheritance edges.

Table 5 gives the breakdown of inheritance edges into these three kinds in our hierarchies. With close to two-thirds of all edges, non-virtual inheritance dominates the two other categories.

Hierarchy	Non-virtual		Virtual		Potentially virtual	
	m_n[i]	$\frac{m_n}{m}$[ii]	m_v[iii]	$\frac{m_v}{m}$[iv]	m_{pv}[v]	$\frac{m_{pv}}{m}$[vi]
Unidraw	471	98.9%	5	1.1%	0	0%
Self	1627	88.5%	6	0.3%	205	11.2%
Laure	268	85.1%	30	9.5%	17	5.4%
Eiffel4	1709	63.8%	464	17.3%	505	18.9%
Ed	423	56.4%	135	18.0%	192	25.6%
LOV	423	54.7%	140	18.1%	211	27.3%
Geode	1305	46.9%	763	27.4%	717	25.7%
Total	6226	64.7%	1543	16.0%	1847	19.2%

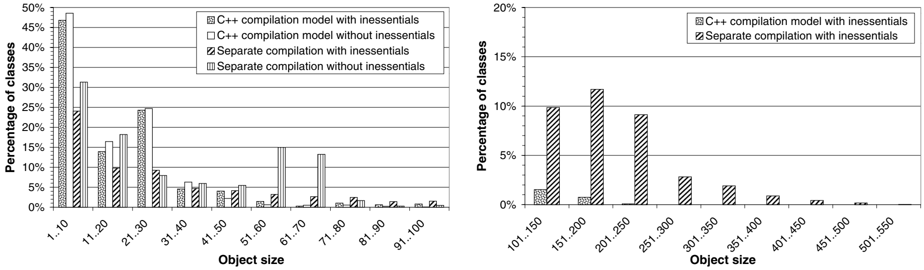
[i] Number of non-virtual edges
[ii] Fraction of non-virtual edges
[iii] Number of virtual edges
[iv] Fraction of virtual edges
[v] Number of potentially virtual edges
[vi] Fraction of virtual edges

Table 5: The kinds of inheritance edges.

There are around 20% of potentially virtual edges, that will be devirtualized in WHO. The number of virtual inheritance, is the smallest and stands at 16%. These edges are the main target of our inlining optimization technique. It is also interesting to note that non-virtual inheritance decreases and virtual inheritances increases, as α increases. On the other hand, there seems to be no direct relationship between α and the fraction of potential-virtual edges.

4.2 Object Size in Different Compilation Strategies

For brevity, we will use the term *object size* to refer to the total number of compiler generated fields in the layout of objects of a certain class. No confusion will arise since our database measurements do not include information on the other kinds of fields stored in an object. Another slight abuse of terminology will be in the use of "*class size*" in the meaning "number of compiler generated fields in objects of the class".



(a) object size 1 . . . 100 (b) object size 101 . . . 550

Fig. 5: Distribution of object size in SEP$^+$, SEP$^-$, CC$^+$, and CC$^-$.

Fig. 5 show how object size values are distributed in the four main strategies: CC^+, CC^-, SEP^+, SEP^-. The X-axis gives the object size in terms of the number of compiler generated fields. The Y-axis is the percentage of classes out of the total of all classes in our database with the corresponding size.

In CC^+ and CC^-, the object size of most classes is under 40. In particular, CC^-, doesn't produce objects greater than 100 in size. Around 2.5% of all classes in CC^+ give an object size in the range of 100 to 250. In SEP, the percentage of mammoth objects begins to grow. Around 30% of objects have size between 50 and 70 in SEP^-. In SEP^+ around 35% of objects will have size between 100 and 550.

With the objective of showing that both time and memory efficient object layout is possible in C++, we choose CC^+ as the *baseline* for benchmarking optimization techniques. Henceforth, unless clearly specified otherwise, all results are presented in comparison to this strategy. Note that this choice of a baseline sets the bar at a pretty high level, since the size of 50% of all classes is 10 or less.

Fig. 6 examines the distribution of object size in CC^+ more carefully.

We see that there are three main peaks to the distribution– at object size 1–5, at around 20, and at around 50. In addition, a smaller and almost equal number of classes can be found at all object sizes in the range 20–150.

In total we see that there is a very significant portion of very small classes: 30% of all classes have at most three compiler generated fields. Moreover, the size of 85% of all classes have no more than 30 compiler generated fields.

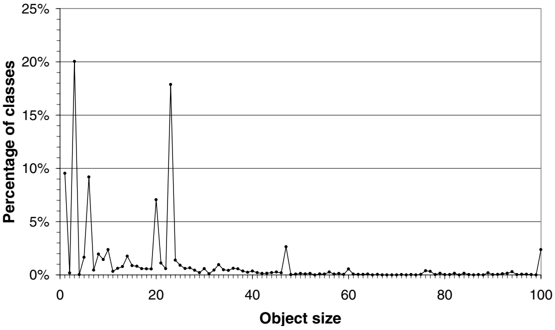


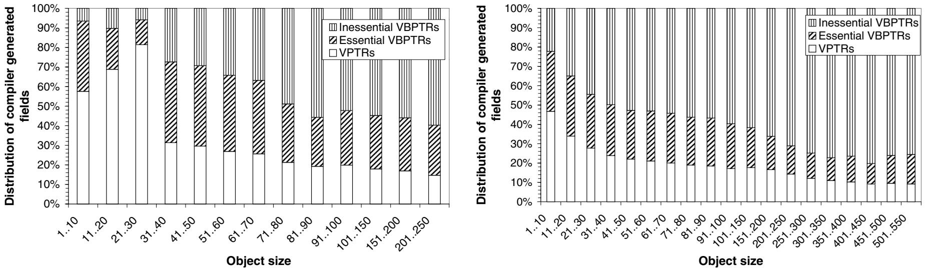
Fig. 6: Distribution of object size in CC^+.

4.3 Distribution of Compiler Generated Fields

As explained before, there are three kinds of compiler generated fields: VPTRS, *e*-VBPTRS, and *i*-VBPTRS. Fig. 7 shows how the total overhead is broken down into these three kinds for different object sizes.

Even though the graph in Fig. 7(a) shows the values for the CC^+ strategy, it is easy to infer from it the distribution in CC^-, which is the same as CC^+, except that no *i*-VBPTRS occur. It is important to note that in the bulk of the objects, i.e., objects of size no greater through 30, 50–80% of the overhead is due to VPTRS. This is indicative of the potential for savings by bidirectional layout techniques of classes of this size. To reduce the size of larger classes, inline techniques must be activated.

A similar breakdown for SEP^+ is shown in Fig. 7(b). Since in SEP^+ all edges are marked virtual, we have that for every VPTR in a subobject, there is at least one *e*-VBPTR in the object or in a containing subobject pointing to this subobject. In this strategy a subobject corresponding to a virtual base will even have two or more VBPTRS pointing



(a) in CC$^+$

(b) in SEP$^+$

Fig. 7: Distribution of compiler generated fields.

to it. The number of i-VBPTRS could be even greater. Therefore, the fact that the majority of the overhead in this strategy is due to VBPTRS is not surprising.

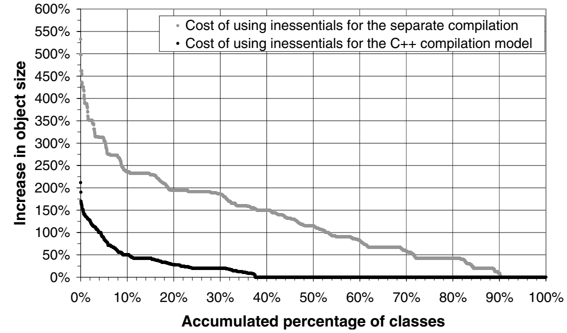


Fig. 8: Accumulative distribution of cost of using inessentials for CC and SEP.

Fig. 8 sheds light from a different angle on the cost of including i-VBPTRS in SEP and in CC. This graph is *accumulative* in the sense that a data point (x, y) means that x percent of all classes experience an increase of y percent or more in size due to i-VBPTRS. More accumulative graphs will follow.

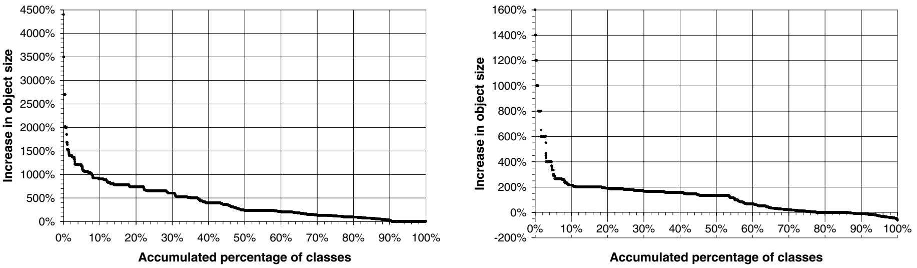
In CC, we see that although there are some classes whose size is almost tripled by i-VBPTRS, over 60% of all classes do not have i-VBPTRS at all, while there are 20% of the classes whose size is increased due to i-VBPTRS by more than 30%. Thus, in the majority of classes, i-VBPTRS induce only a modest increase to object size. This gives a good justification to the decision of numerous compiler writers to use i-VBPTRS in C++ compilers. However, since applications may instantiate classes at varying frequencies, there should be a strong effort to optimize the use of i-VBPTRS in those 10% of classes in which the increase is 50% or more. In contrast, the use of i-VBPTRS in the SEP model may put a heavy strain on resources. The size of 50% of all classes is at least doubled in switching from SEP$^-$ to SEP$^+$.

4.4 Cost of Separate Compilation

It is quite apparent from Fig. 5 that SEP generates much larger objects than CC. It is however interesting to compare the size of the same objects in these two different strategies.

Fig. 9(a) shows the increase in object size when SEP$^+$ is chosen over CC$^+$. With close to 40% of classes suffering a six-fold increase, and some classes increasing by a factor of over forty, the CC$^+$ strategy seems to be truly impractical. In a separate compilation model the decision to store *i*-VBPTRs should not be made lightly.

The alternative, namely the SEP$^-$ strategy, incurs a cost of an increase in the time to access indirect virtual bases.



(a) choosing SEP$^+$ over CC$^+$

(b) choosing SEP$^-$ over CC$^+$

Fig. 9: Accumulative distribution of increase in object size.

Let us now compare SEP$^-$, the more memory-efficient version of SEP, with CC$^+$, the memory-wasteful version of CC, as done in Fig. 9(b). Although there is a small fraction (less than 10%) of classes in which SEP$^-$ is more efficient than CC$^+$, in the majority of classes CC$^+$ is much more efficient than SEP$^-$. Phrased differently, C++ without the virtual annotation by user, and without whole program analysis will incur a median increase in object size of over 100%!

5 Eliminating Transitive Inheritance

ETRANS is a preliminary step to more advanced optimization techniques, in which all transitive virtual inheritance edges are eliminated. Since by assumption there is no repeated inheritance in our hierarchies, this step is basically to remove *all* transitive edges. Beyond the obvious simplification of the hierarchy graph, the elimination of transitive virtual edges reduces object size. ETRANS can be also implemented in SEP, since even in separate compilation it is assumed that whenever a class is compiled, information on all of its ancestors is available.

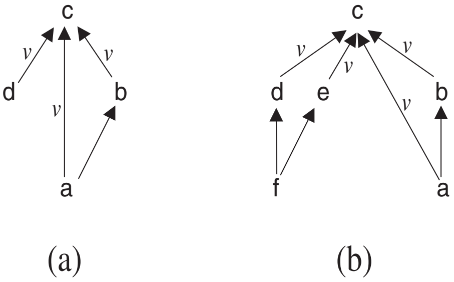


Fig. 10: Change in status of edges due to ETRANS.

We will use the name of a compilation strategies as a subscript in a name of an optimization technique to emphasize the environment where the technique is applied, for instance ETRANS$_{CC^+}$ will notify ETRANS applied in CC$^+$ environment.

ETRANS reduces object size due to the fact that transitive edges, by definition, make diamonds hierarchies (Fig. 1(b)). Typically, an elimination of a transitive edge will also eliminate a diamond. This could make virtual edges into potentially virtual, or even non-virtual. In addition, potentially virtual edges may become non-virtual.

Consider for example the hierarchy of Fig. 10(a). By eliminating the transitive edge (a, c), the edge (b, c) changes its status from virtual to non-virtual.

Also, the edge (d, c) changes its status from potentially virtual to non-virtual. Removing the transitive edge (a, c) in the hierarchy of Fig. 10(b), changes the status of the edge (b, c) from virtual to potentially virtual.

Table 6 describes the changes in distribution of the inheritance edges by kind due to ETRANS$_{CC+}$. The fact that as many as 4.1% of all edges were eliminated indicates that programmers do make use of transitive inheritance edges, probably as a matter of design style, since transitive inheritance contributes only very little to the semantics. It is evident however that there are hierarchies in which no transitive edges were used, although at least in C++ this is not forbidden by the programming language.

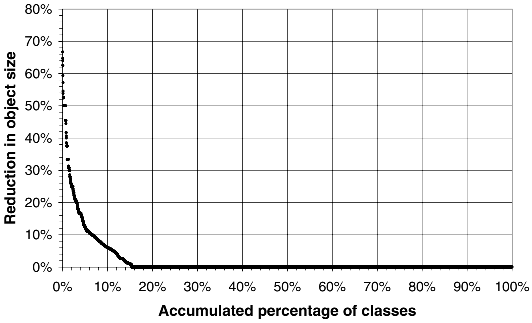


Fig. 11: Accumulative distribution of efficacy of ETRANS$_{CC+}$.

Hierarchy	Δm[(i)]	Non-virtual $\frac{m_n}{m}$[(ii)]	Non-virtual Δm_n[(iii)]	Virtual $\frac{m_v}{m}$[(iv)]	Virtual Δm_v[(v)]	Potentially virtual $\frac{m_{pv}}{m}$[(vi)]	Potentially virtual Δm_{pv}[(vii)]
Unidraw	0.0%	98.9%	0.0%	1.1%	0.0%	0.0%	0.0%
Self	0.0%	88.5%	0.0%	0.3%	0.0%	11.2%	0.0%
Laure	0.0%	85.1%	0.0%	9.5%	0.0%	5.4%	0.0%
Eiffel4	-1.6%	65.8%	+1.5%	15.6%	-11.4%	18.6%	-2.8%
Ed	-3.9%	70.3%	+19.9%	13.6%	-27.4%	16.1%	-39.6%
LOV	-3.5%	61.8%	+9.2%	14.3%	-23.6%	23.8%	-15.6%
Geode	-10.7%	60.7%	+15.6%	16.8%	-45.3%	22.6%	-21.8%
Total	-4.1%	71.3%	+5.6%	11.6%	-30.4%	17.0%	-15.1%

[(i)] Relative change in the total number of edges
[(ii)] New fraction of non-virtual edges
[(iii)] Relative change in the number of non-virtual edges
[(iv)] New fraction of virtual edges
[(v)] Relative change in the number of virtual edges
[(vi)] New fraction of potentially virtual edges
[(vii)] Relative change in the number of potentially virtual edges

Table 6: The impact of ETRANS$_{CC+}$ on the distribution of inheritance edges.

About one in three virtual edges was either eliminated or made into non-virtual or potentially virtual. This brought the percentage of virtual edges from 16% (Table 5) down to 11.6%. The reduction in the percentage of potentially virtual edges was more modest (17% instead of 19.2%). The most dramatic changes in the distribution were of course in Geode since this hierarchy had the largest fraction of transitive edges.

In order to describe the impact of an optimization technique on the object size we will use the term *efficacy*.

Definition 1. *The* efficacy *of optimization technique for a certain class is the relative reduction in object size of a class due to application of the technique.*

Fig. 11 shows the efficacy of ETRANS$_{CC^+}$. As can be seen in the figure, ETRANS is not a particularly effective optimization technique. Even though there is a very small fraction of classes which enjoy a significant reduction of 40% or more, overall only 8% of all classes experience a saving of 8% or more in their size.

6 Inlining Techniques

Consider the class hierarchy of Fig. 12. In this hierarchy, classes a and e are virtual bases and edges (b, a), (c, a), (d, a), (e, a) (h, e) and (i, e) are all virtual. Also, (f, a) is designated as potentially virtual in the CC strategy since it is incident on a virtual base. A whole program analysis will reveal that (f, a) is not virtual and will *devirtualize* it.

After DEVIRT was applied, S-INLN optimization, chooses exactly one of virtual edges incident on a virtual base, and inlines this base down this edge. In this example, a virtual base a may be inlined into any one of *b*, *c*, *d* or *e* by S-INLN. A-INLN improves on S-INLN by choosing a *maximal* subset of virtual edges incident on a virtual base, such that this virtual base can be inlined into all of them. A-INLN will inline a into b, e and f, which is the maximal set of children of a, such that no two of them share a descendant.

Note that devirtualization and the other inlining optimization techniques ([11]), are greatly simplified thanks to the absence of duplicate inheritance. Class a could not be inlined into e under no circumstances, if k also had a repeated inheritance link to e.

Every time inlining takes place, an *e*-VBPTR is eliminated. Inlining also has the potential of eliminating *i*-VBPTRS since it cuts by one the number of dereference operations of *e*-VBPTR required to reach a virtual base.

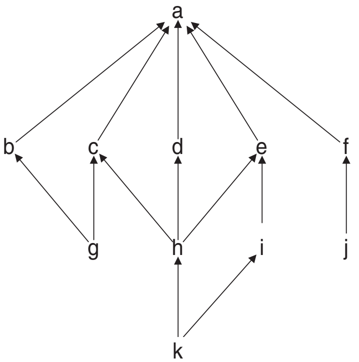


Fig. 12: A class hierarchy demonstrating different levels of inlining.

Since (e, a) and (i, e) are virtual edges, the *i*-VBPTR leading from i to a can be eliminated as soon as a is inlined into e. Inlining has also the potential to eliminate VPTRS. In the example, when a is inlined into f, the VPTR of a f object can be shared with the VPTR of its a subobject.

6.1 The Extent of Inlining

Table 7 gives the fraction of edges which are inlined by each of these three techniques.

Hierarchy	DEVIRT	SINLN_{CC+}	AINLN_{CC+}
Unidraw	0.0%	0.4%	0.4%
Self	11.2%	0.2%	0.2%
Laure	5.4%	3.5%	5.1%
Eiffel4	18.6%	3.5%	9.5%
Ed	16.1%	2.4%	6.0%
LOV	23.8%	2.8%	6.3%
Geode	22.6%	3.1%	8.6%
Total	17.0%	2.4%	6.3%

Table 7: The fraction of edges inlined by DEVIRT, SINLN_{CC+} and AINLN_{CC+}.

Comparing this to Table 6 we see that all potentially virtual edges were inlined by DEVIRT. We also see that SINLN_{CC+} inlines only 2.4% of all edges, i.e., less than one in four of all virtual edges. There are significant differences in the extent of inlining of SINLN_{CC+}. Two out of three virtual edges are inlined in Self, while less than one in five edges are inlined in Geode. This variance and the small extent of SINLN_{CC+} can be explained by the huge standard deviation in the number of children (column β in Table 4). Since S-INLN inlines into only one descendant, its impact would be smaller in all virtual bases with large number of descendants.

Note that the variance between the hierarchies in the extent of inlining by AINLN_{CC+} is much smaller than that of SINLN_{CC+}. Overall, AINLN_{CC+} is successful in inlining more than one in two virtual edges.

6.2 Implementation of Aggressive Inlining

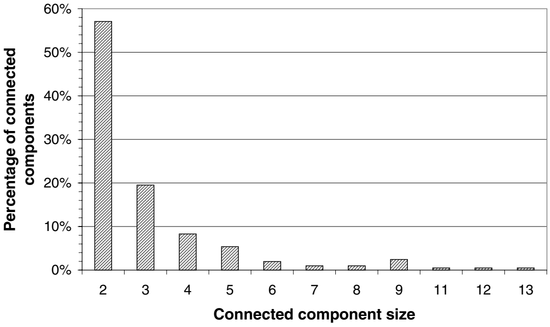
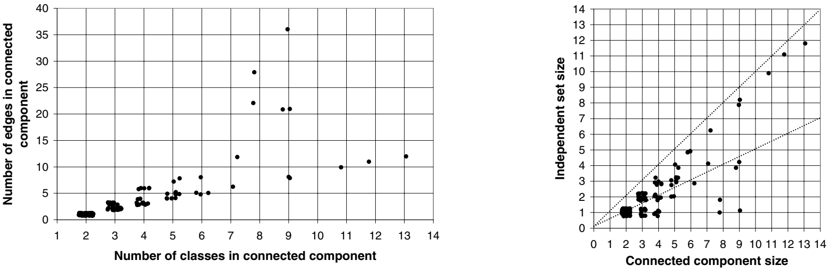


Fig. 13: Distribution of size of connected components.

As we said, A-INLN uses a maximal independent set algorithm. Two immediately derived classes of a virtual base are dependent if they share a descendant, because the virtual base must occur exactly once in that descendant. For each virtual base A-INLN creates a dependency graph of all of its immediately derived classes. This graph is then broken into its connected components, and the maximally independent set of nodes in each component is then found. We found that the vast majority (98%) of components are small, having no more than 13 nodes. For such small graphs, an exhaustive search algorithm with minor optimizations can be used to find the maximally independent set.

From Fig. 13 describing the distribution of size of small connected components, we see that close to 90% of all connected components have no more than 4 nodes—which makes the exhaustive search approach even more appealing.

Fig. 14(a) is a scatter diagram giving the number of edges in those components. Since the coordinates of all points are integral and drawn from a small range, many data points may coincide at the same grid location. To visualize these clusterings, we add a small random *perturbation* $-0.25 \leq \epsilon \leq 0.25$ to all values. This technique is used in all subsequence scatter diagrams as well. We see that most of the components are sparse. In fact, 82% of all components are trees. Now, since a tree is also a bipartite graph, the maximally independent set has at least half of the nodes. Fig. 14(b) describes the size of the maximal independent sets found in small connected components. The two diagonal guidelines drawn on the grid show that in the maximal independent set is at least half in size of most connected components. In fact, there are many connected components in which all but one node take part in the maximal independent set.



(a) Distribution of edges and nodes

(b) Distribution of size of maximal independent sets

Fig. 14: Small connected components.

6.3 The Efficacy of Inlining Algorithms

Even though the extent of inlining is interesting, it is much more important to determine how much inlining reduces object size.

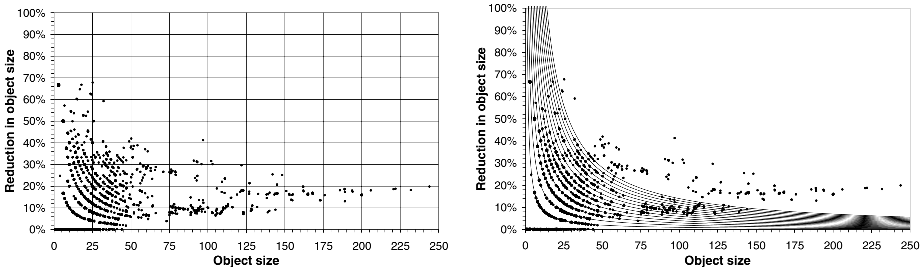
For the purpose of presentation in this section we will include DEVIRT in SINLN$_{CC^+}$ and in AINLN$_{CC^+}$. Thus, SINLN$_{CC^+}$ will be at least as good as DEVIRT and AINLN$_{CC^+}$ will be at least as good as SINLN$_{CC^+}$.

Fig. 15(a) is a scatter diagram showing the reduction in object size due to DEVIRT. It is important to note that DEVIRT reduces the size of *all* classes with 45 or more compiler generated fields. Also, even though there are small classes in which no savings are made, there is a cluster of large classes of size 50 or more with savings in the range of 10-20%.

A somewhat closer look at Fig. 15(a) reveals that the points appear to lie on hyperbolic curves. This is by no means a coincidence!

A saving of f fields in a class of size b will place a point $(b, f/b)$ on the graph. In other words, all classes with a saving of f fields can be found on the hyperbola $y = f/x$. To illustrate this, we redraw Fig. 15(a) in Fig. 15(b) with the hyperbolic curves corresponding to $f = 1, \ldots, 14$.

Fig. 16(a) is a similar scatter diagram for SINLN$_{CC^+}$. This time we see that savings occur in all classes whose size is greater than 30. Savings in large classes with more than 50 compiler generated fields is clustered in the 25%-30% range.

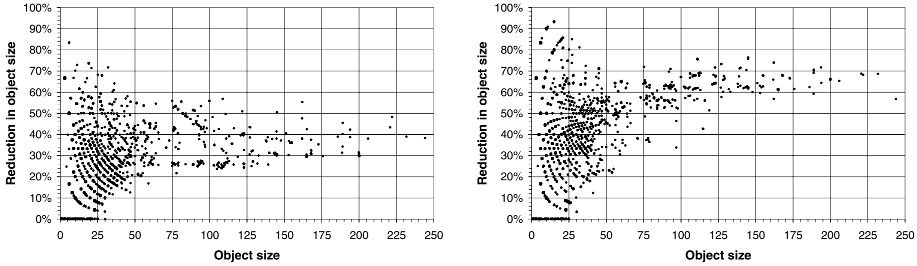


(a) Efficacy of DEVIRT

(b) Fig. 15(a) with the first 14 hyperbolic curves

Fig. 15: Eifficacy of DEVIRT.

As can be seen from Fig. 16(b) AINLN$_{CC^+}$ guarantees savings at some level for all classes whose size is greater than 25. This time, the typical saving in large classes is around 60%, and even more than that for really large classes.



(a) SINLN$_{CC^+}$

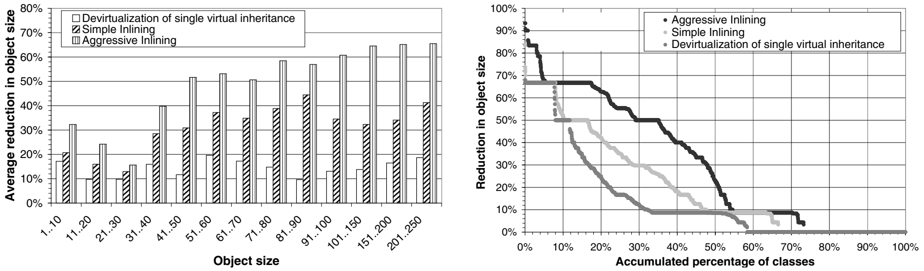
(b) AINLN$_{CC^+}$

Fig. 16: Efficacy of inlining techniques in CC$^+$.

The three inlining techniques are compared in Fig. 17(a). We see that DEVIRT gives a consistent average saving of 10–20% in all object sizes. As explained earlier, WHO is nothing but CC after DEVIRT is applied. We argue that WHO$^+$ is much like CC$^+$, except that objects tend to be 10–20% smaller. Such saving may not seem enough to warrant transition to whole program analysis. Perhaps this is why many widely available C++ compilers do not apply whole program analysis to reduce object size. The shift to WHO can however be justified by the efficacy of the advanced optimization techniques.

It is important to note that whole program analysis for object layout does not mean in C++ (say) a batch compile of the whole program. All that is required is that object layout is decided in link time, and that appropriate fixup table is used to patch object-code files.

In small classes with no more than 30 compiler generated fields, SINLN$_{CC^+}$ does not add much savings to DEVIRT. In larger classes, the additional savings by SINLN$_{CC^+}$ are in the 15–35% range.



(a) Average efficacy of inlining techniques

(b) Accumulative distribution of efficacy of inlining techniques

Fig. 17: Average and accumulative efficacy of inlining in CC$^+$.

AINLN$_{CC^+}$ is significantly better than DEVIRT and SINLN$_{CC^+}$ in all object sizes, with the exception of classes whose size lies in the 21–30 range. This is unfortunate, since as observed in Sec. 4.2 one of the peaks in the distribution of classes lies exactly in this range. However, as we shall see below, bidirectional layout optimization techniques will be particularly effective in this range.

Another perspective at the comparison of the three inlining techniques is offered by Fig. 17(b). From the figure we can see for example that AINLN$_{CC^+}$ effects a 50% saving or more in 35% of all classes, SINLN$_{CC^+}$ makes a saving of at least 30% in 30% of all classes, while DEVIRT saves at least 25% of object size in 20% of all classes.

7 Bidirectional Object Layout

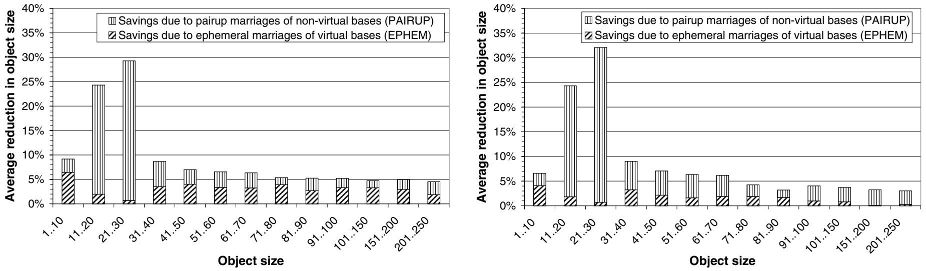
The bidirectional object layout idea is that all root classes and all other classes for which this is possible, are assigned a *directionality*. The directionality, which could be either positive or negative, is selected using a two-wise independent hash function. Since both positive and negative classes have their VPTR at offset zero, it is possible to "marry" a positive and negative bases in a derived class, whereby saving a VPTR. For example, in Fig. 1(a), if b is negative and c is positive, then a marriage could occur in a. Class a would then have *mixed* directionality. No VBPTRS can be saved by this technique.

7.1 Applying Bidirectional Layout after Inlining

Fig. 19(a) is a scatter diagram showing the reduction in object size due to BIDIR applied after A-INLN was applied.

At first look, it appears that BIDIR$_{CC^+}$ is not a very effective technique, since, asymptotically, for large classes, it gives only about 5% reduction in object size. This however should be expected, recalling that BIDIR$_{CC^+}$ can only eliminate VPTRS. As we saw in Fig. 7(a) on average less than 20% of the size of classes whose size is greater than 80 is used by VPTRS. Thus, the $\approx$ 5% observed saving is not in total disagreement with the theoretical prediction ([11]) of saving of around a quarter of VPTRS.

BIDIR$_{CC^+}$ has a stronger impact on smaller classes in which the weight of VPTRS is greater. We see in Fig. 19(a) that there is a cluster of classes, sized 20–25, in which the



(a) PAIRUP /SINLN$_{CC+}$ and EPHEM /SINLN$_{CC+}$

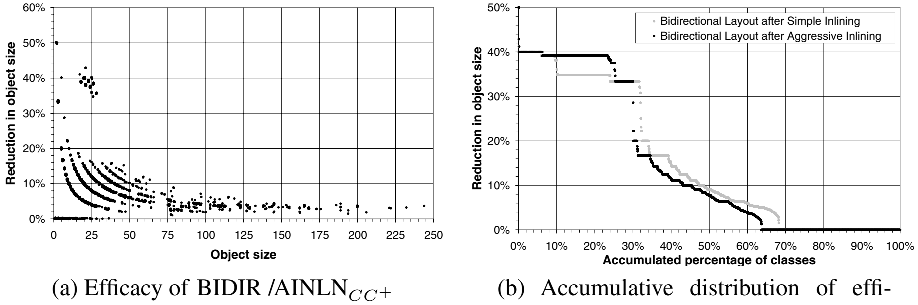
(b) PAIRUP /AINLN$_{CC+}$ and EPHEM /AINLN$_{CC+}$

Fig. 18: Average efficacy of PAIRUP and EPHEM.

saving in object size is around 40%. Theses findings are confirmed by Fig. 18(b) shows the average savings due to BIDIR$_{CC+}$ in different object sizes. We see that for the peak in the distribution of class size located at the 21–30 bracket, BIDIR$_{CC+}$ makes a hefty saving of over 30%.

BIDIR is in fact a combination of two optimization techniques: PAIRUP, in which inlined and non-virtual parents are persistently married, and EPHEM in which virtual bases are ephemerally married. Fig. 18(b) shows also how the savings are broken down between these two sub-techniques.

It is quite surprising to see that even in large classes, not too much saving is owed to EPHEM. This is quite surprising in view of Fig. 3 which shows that there are many classes with a large number of virtual bases. The explanation is that the preliminary inlining optimization eliminated so many virtual bases that the optimization opportunities of EPHEM were severely restricted. This phenomena can also be seen by examining Fig. 18(a), which shows how the saving due to BIDIR are broken down between PAIRUP and EPHEM after SINLN$_{CC+}$, a less effective inlining technique.



(a) Efficacy of BIDIR /AINLN$_{CC+}$

(b) Accumulative distribution of efficacy of BIDIR /SINLN$_{CC+}$ and BIDIR /AINLN$_{CC+}$

Fig. 19: Efficacy and accumulative distribution of BIDIR /SINLN$_{CC+}$ and BIDIR /AINLN$_{CC+}$.

In comparing Fig. 18(a) and (b), we see that although BIDIR after S-INLN is slightly better than BIDIR after A-INLN, their performance is quite similar. However, when S-INLN was applied, EPHEM took a greater share of the total achievements of BIDIR.

Yet another comparison of between these two alternatives is given in Fig. 19(b). Again, we see that the two alternatives are quite similar: they both achieve about 30% savings in about 30% of all classes. The conclusion from this data is that inlining may hinder bidirectional optimization it is better to apply A-INLN and then BIDIR. This conclusion is confirmed by theoretical, probabilistic considerations which show that the expectation of the saving in VPTRs is decreased by inlining.

7.2 Hermaphrodite Object Layout

In plain bidirectional layout marriage opportunities may be missed due to unfortunate selection in the pseudo-random assignment of directionality to classes.

Going back again to Fig. 1(a), if the hash function assigned a positive directionality to both b and c, then no VBPTR could be saved in a. *Hermaphroditing* is an object optimization technique designed to overcome this problem.

A *hermaphrodite* class is a class whose instances may use one of two different layouts: one using positive and the other with negative one. Initially, all root classes are hermaphrodite. When a class u inherits from h hermaphrodite parents, then if h is even, $h/2$ marriages occur, where appropriate directionalities are selected for each of the h subobjects.

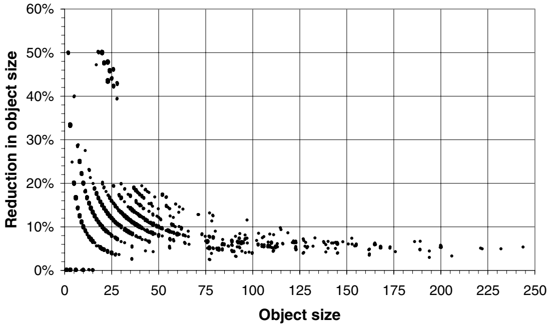


Fig. 20: Efficacy of H-BIDIR /AINLN$_{CC+}$.

The class u has then mixed directionality, but there is only one possible layout for its objects. If h is odd, then $(h - 1)/2$ marriages occur, fixing the directionality of $h - 1$ subobjects. The remaining parent, v, remains hermaphrodite in u. Class u becomes hermaphrodite as well, where its instances of positive (respectively negative) directionality use a positively (negatively) directed u subobject.

All hermaphrodite classes have their VPTR at offset zero. However, in order to make a virtual function call, it must be possible to determine at run time the directionality of a specific instance, since the directionality of the VTBL is the same as the directionality of the object.[4] Since in many modern architectures, not all bits of a pointer are used, directionality could be stored as one of these bits, say LSB. Here is the pseudo-code for invoking a virtual function f using a pointer p to a hermaphrodite class.

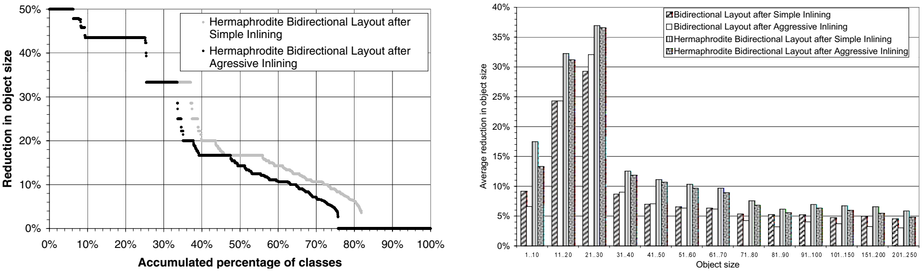
The implementation of this algorithm in machine code and its run time efficiency are very dependent on the instruction set. It could even be possible to develop dedicated hardware support. Conversely, with the absence of such support, this dispatch mechanism would bloat the code significantly. All these questions are beyond the scope of this paper. Instead, we are interested in the saving in per-object memory that this technique may

[4] A similar need arises for data members access. However, data members access could be restricted to member functions, and each hermaphrodite object could have also two versions for each of its member functions.

bring about. Fig. 20 shows the saving in object size due to H-BIDIR when applied after AINLN$_{CC^+}$. The results are similar in appearance to Fig. 19(a), except that as expected, greater savings are demonstrated in all object sizes. For example, the cluster of classes of size 20–25 sees a 45% reduction in its size rather than 40%.

Fig. 21(a) compare the performance of HBIDIR$_{CC^+}$ relative to the inlining algorithm that was applied prior to it. It appears, and in a much clearer way than in Fig. 19(b) that better inlining has adverse affects on the performance of a bidirectional layout. For example, if SINLN$_{CC^+}$ was applied than 37% of all classes see a reduction of a third or more in their size, whereas only 33% of all classes see a similar reduction if AINLN$_{CC^+}$ was applied.

```
VTBL $\leftarrow *\lfloor p/2 \rfloor$
off $\leftarrow$ offset($f$)
**if** odd($p$) **then**
    off $\leftarrow -$off
**end if**
call VTBL [off]
```



(a) Accumulative distribution of efficacy of H-BIDIR /SINLN$_{CC^+}$ and H-BIDIR /AINLN$_{CC^+}$

(b) Average efficacy of BIDIR /SINLN$_{CC^+}$, BIDIR /AINLN$_{CC^+}$, H-BIDIR /SINLN$_{CC^+}$ and H-BIDIR /AINLN$_{CC^+}$

Fig. 21: Accumulative and average efficacy of BIDIR and H-BIDIR in CC$^+$.

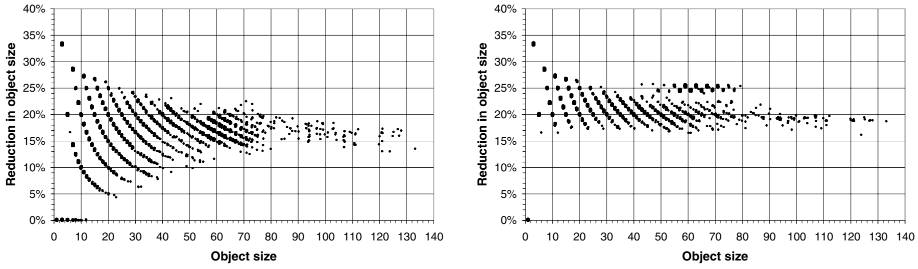
Finally, Fig. 21(b) compares the performance of BIDIR$_{CC^+}$ and HBIDIR$_{CC^+}$ when applied after SINLN$_{CC^+}$ and AINLN$_{CC^+}$, Even though HBIDIR$_{CC^+}$ is better than BIDIR$_{CC^+}$, the differences are not so big, except for the smaller classes, sized 1–10. The incurred dispatch cost of H-BIDIR could be justified in those applications which have a large number of classes of this kind, which is hard to optimize using any other technique.

7.3 Using Bidirectional Optimization in Separate Compilation Model

In contrast with inlining optimization, bidirectional object layout can be done without whole program analysis. It is therefore interesting to evaluate the efficacy of these algorithms in a separate compilation model. Since SEP$^+$ seems to be impractical, SEP$^-$ is used as baseline for this benchmarking.

The savings in object size due to BIDIR$_{SEP^-}$ in are shown in Fig. 22(a). The figure demonstrates that there is a cluster of classes with savings in the 15–20% range. Further, saving is guaranteed for all classes with 13 or more compiler generated fields.

As Fig. 22(b) shows, the switch to hermaphroditing in SEP$^-$ raises a bit the range of typical savings.



(a) BIDIR$_{SEP-}$

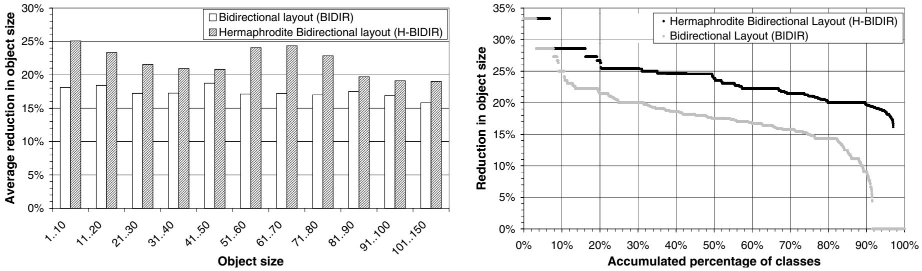
(b) HBIDIR$_{SEP-}$

Fig. 22: Efficacy of Bidirection Layout techniques in SEP$^-$.

Moreover, the figure demonstrates what can also be easily inferred from a pigeon hole principle: saving is guaranteed for all classes of size at least three.

Fig. 23(a) gives a side by side comparison of the efficacy of BIDIR and H-BIDIR in SEP$^-$ in SEP$^-$ for different object sizes. In no range the efficacy of H-BIDIR is bounded above by 25%. This bound is explained by our previous observation that in SEP$^-$, the number of VBPTRS is about the same as that of VPTRS, no savings in VBPTRS are made by bidirectional techniques, whereas at most half of all VPTRS can be saved. This meager saving of quarter in object size is far from bridging the gap between CC$^-$ and SEP$^+$ (Fig. 9)(a).

It is also worth noting that in no size bracket HBIDIR$_{SEP-}$ is better than BIDIR$_{SEP-}$ by more than 8%. The difference between the techniques is also highlighted in Fig. 23(b) showing the distribution of their savings in SEP$^-$.



(a) Average efficacy of BIDIR$_{SEP-}$ and HBIDIR$_{SEP-}$

(b) Accumulative distribution of efficacy of BIDIR$_{SEP-}$ and HBIDIR$_{SEP-}$

Fig. 23: Average and accumulative efficacy of Bidirection Layout techniques in SEP$^-$.

8 Summary

Table 8 summarizes the efficacy of the optimization techniques introduced in this paper. With aggressive lining we can achieve average savings of 32%, regardless whether the average is computed over all classes or just leaves. Applying bidirectional layout after inlining gives an additional average saving of 14.7%, which is very close to the efficacy of devirtualization. In applying the whole optimization suit, almost 50% of compiler generated fields can be eliminated. The last two lines in Table 8 are concerned with SEP$^-$. Bidirectional object layout optimization reduces object size in this case by more than 17%.

	Savings			
	Median		Average	
Technique	overall	leaves	overall	leaves
CC$^+$				
ETRANS	0.0%	0.0%	2.0%	1.9%
DEVIRT	8.5%	8.7%	13.9%	16.3%
S-INLN	0.0%	0.0%	6.0%	5.5%
A-INLN	0.0%	0.0%	16.0%	13.9%
BIDIR /S-INLN	9.1%	8.6%	15.2%	15.0%
BIDIR /A-INLN	7.7%	7.1%	14.7%	14.7%
H-BIDIR /S-INLN	16.7%	16.7%	21.8%	20.8%
H-BIDIR /A-INLN	14.3%	13.6%	19.7%	19.5%
SEP$^-$				
BIDIR	17.5%	17.9%	17.2%	17.4%
H-BIDIR	23.8%	23.5%	23.4%	23.3%

Table 8: The efficacy of all optimization techniques.

9 Conclusions

We studied the topological properties of large inheritance hierarchies which make use of MI. It was found that not very many classes make direct use of MI, and the ones that do have a relatively small number of parents. Further, the number of virtual bases is relatively small. Nonetheless, we found that the effects of MI can be seen throughout the hierarchy, and many classes have a large number of virtual bases. Despite this, hierarchy DAGs tend to share many of the properties of balanced trees.

The object size due to several layout strategies was investigated. Large classes with 50 and more compiler generated fields were found in significant portions even for CC$^-$, which seems to be the most efficient strategy in wide use. Even larger classes were found in CC$^+$, the strategy favored by most C++ compilers. Object sizes found in SEP$^+$ seemed so large to render this strategy impractical. On the other hand, SEP$^-$ may suffer from timing problems due to the cost of casting to virtual bases, although we were unable to give an accurate estimate on the penalty in runtime in this strategy.

Concentrating on CC^+, we studied a variety of optimization techniques, grouped into two families: inlining and bidirectional layout. It was found that inlining is particularly effective for large classes, while bidirectional layout is more suitable for medium and small sized objects.

The use of hermaphrodite object layout, a new optimization technique contributed here, can improve on bidirectional layout by another 5–6% in general. However, hermaphroditing is especially effective in very small classes, where it guarantees saving in all classes having more than one VPTR or more than one *e*-VBPTR. It is not clear however what is the runtime cost of dispatch with hermaphrodite layout.

Some of the directions for future research include:

Repeated inheritance. Based on several independent reasons we assumed here that no repeated inheritance can occur. It would be interesting to study the extent to which repeated inheritance is used in practice, and if indeed it is found that there is more an occasional use of it, to try to optimize it. The main difficulty is that bidirectional layout is much less effective in the presence of repeated inheritance, and that inlining is only effective for virtual inheritance.

Instantiation frequency. Even though our working assumption was that classes are equally likely to be instantiated, the efficacy of optimization techniques was also studied for different object size. Sweeney and Burke [28] confirmed that there is a wide variety in the frequency of instantiation of different classes. It would be important to experiment with these optimization techniques in a sample of applications running with a sample of inputs.

Data member optimization. Non-compiler generated fields were not included in our study. It is important not only to compare the overhead due to compiler generated fields with actual object size, but also to apply our techniques to the optimization of ordinary data members. For example, bidirectional layout can easily be applied to expanded fields. For fields which use reference semantics one might be able to use concepts of ownership [21] and Balloon types [2] for the purpose of inlining.

Acknowledgments. We are grateful to Peter Sweeney of the IBM T.J. Watson research center for fruitful, stimulating and encouraging discussions.

References

[1] M. Akşit and S. Matsuoka, editors. *Proceedings of the 11th European Conference on Object-Oriented Programming*, number 1241 in Lecture Notes in Computer Science, Jyväskylä, Finland, June 9-13 1997. ECOOP'97, Springer Verlag.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In Akşit and Matsuoka [1].

[3] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, 1996.

[4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual calls. In OOPSLA'96 [22].

[5] T. Cargill, B. Cox, W. Cook, M. Loomis, and A. Snyder. Is multiple inheritance essential to OOP? Panel discussion at the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95) (Washington, DC), Oct. 1993.

[6] Y. Caseau. An object-oriented deductive language. *Annals of Mathematics and Artificial Intelligence*, Mar. 1991. Special issue on deductive databases.

[7] Y. Caseau. Efficient handling of multiple inheritance hierarchies. In Paepcke [23], pages 271–287.

[8] B. J. Cox. *Object-Oriented Programming - An Evolutionary Approach.* Addison-Wesley, 1986.

[9] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *Proceedings of the 10th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 141–155, Austin, Texas, USA, Oct. 15-19 1995. OOPSLA'95, Acm SIGPLAN Notices 30(10) Oct. 1995.

[10] K. Driesen and U. Hölzle. The direct cost of virtual functions calls in C++. In OOPSLA'96 [22], pages 306–323.

[11] J. Y. Gil and P. Sweeney. Space- and time-efficient memory layout for multiple inheritance. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 256–275, Denver, Colorado, Nov.1-5 1999. OOPSLA'99, Acm SIGPLAN Notices 30(11) Nov. 1999.

[12] R. Giladi and N. Ahituv. SPEC as a pefromance evaluation measure. *Computer*, 28(8):33–42, Aug. 1995.

[13] R. Godin and H. Mili. Building and maintaining analysis-level class hierarchies using galois lattices. In Paepcke [23], pages 394–410.

[14] Interactive Software Engineering. Ise eiffel compiler. See `http://www.eiffel.com`, 1999.

[15] A. Krall, J. Vitek, and R. N. Horspool. Efficient type inclusion tests. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 142–157, Atlanta, Georgia, Oct. 5-9 1997. OOPSLA'97, Acm SIGPLAN Notices 32(10) Oct. 1997.

[16] A. Krall, J. Vitek, and R. N. Horspool. Near optimal hierarchical encoding of types. In Akşit and Matsuoka [1], pages 128–145.

[17] M. A. Linton and D. Z. Pan. Interface translation and implementation filtering. In D. Lea, editor, *the 6th C++ Conference*, pages 227–236, Cambridge, MA, Apr. 1994. USENIX.

[18] B. Magnussun, B. Meyer, and et al. Who needs need multiple inheritance. Panel discussion at the European conference on Technology of Object Oriented Programming (TOOLS Europe'94), Mar. 1994.

[19] B. Meyer. *Object-Oriented Software Construction.* International Series in Computer Science. Prentice-Hall, 1988.

[20] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, second edition, 1997.

[21] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 158–185, Brussels, Belgium, July20–24 1998. ECOOP'98, Springer Verlag.

[22] OOPSLA'96. *Proceedings of the 11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, San Jose, California, Oct. 6-10 1996. Acm SIGPLAN Notices 31(10) Oct. 1996.

[23] A. Paepcke, editor. *Proceedings of the 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Washington, DC, USA, Sept. 26 - Oct. 1 1993. OOPSLA'93, Acm SIGPLAN Notices 28(10) Oct. 1993.

[24] M. Sakkinen. The darker side of C++ revisited. *Structured Programming*, 13:155–177, 1992.

[25] Standard Performance Evaluation Coroporation. SPECjvm98 documentation, release 1.0. Online version at `http://www.spec.org/osg/jvm98/jvm98/doc/index.html`, Aug. 1998.

[26] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, Mar. 1994.

[27] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[28] P. F. Sweeney and M. Burke. A methodology for quantifying and evaluating the space overhead in C++ object models. Technical Report RC21370, IBM T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, USA, Dec. 1998.

[29] D. Ungar and R. B. Smith. SELF: The power of simplicity. In N. K. Meyrowitz, editor, *Proceedings of the 2^{n} Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–241, Orlando, Florida, Oct. 4-8 1987. OOPSLA'87, Acm SIGPLAN Notices 22(12) Dec. 1987.

Optimizing Java Programs in the Presence of Exceptions

Manish Gupta, Jong-Deok Choi, and Michael Hind

IBM Thomas J. Watson Research Center, P.O. Box 704,
Yorktown Heights, NY 10598, USA
{mgupta,jdchoi,hindm}@us.ibm.com

Abstract. The support for precise exceptions in Java, combined with frequent checks for runtime exceptions, leads to severe limitations on the compiler's ability to perform program optimizations that involve reordering of instructions. This paper presents a novel framework that allows a compiler to relax these constraints. We first present an algorithm using dynamic analysis, and a variant using static analysis, to identify the subset of program state that need not be preserved if an exception is thrown. This allows many spurious dependence constraints between potentially excepting instructions (PEIs) and writes into variables to be eliminated. Our dynamic algorithm is particularly suitable for dynamically dispatched methods in object-oriented languages, where static analysis may be quite conservative. We then present the first software-only solution that allows dependence constraints among PEIs to be completely ignored while applying program optimizations, with no need to execute any additional instructions if an exception is not thrown. With a preliminary implementation, we show that for many benchmark programs, a large percentage of methods can be optimized (while honoring the precise exception requirement) without any constraints imposed by frequent runtime exceptions. Finally, we show that relaxing these reordering constraints can lead to substantial improvements (up to a factor of 7 on small codes) in the performance of programs.

1 Introduction

Java [14] continues to gain importance as a popular object-oriented programming language for general-purpose programming. Although some aspects of Java, such as strong typing, simplify the task of program analysis and optimization, other aspects, such as support for precise exceptions, can hamper program analysis and optimizations. The Java language specification requires that exceptions be *precise*, which implies that

1. when an exception is thrown, the program state observable at the entry of the corresponding exception handler must be the same as in the original program; and
2. exception(s) must be thrown in the same order as specified by the original (unoptimized) program.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 422–446, 2000.
© Springer-Verlag Berlin Heidelberg 2000

To satisfy the precise exception requirement, Java compilers disable many important optimizations across instructions that may throw an exception (we refer to these *potentially excepting instructions* as PEIs [19]). This hampers a wide range of program optimizations such as instruction scheduling, instruction selection (across a PEI), loop transformations, and parallelization. Furthermore, PEIs are quite common in Java programs – frequently occurring operations such as reads and writes of instance variables, array loads and stores, method calls, and object allocations may all throw an exception. Hence, the ability of the compiler to perform any program transformation that requires instruction reordering is severely limited, which impedes the performance of Java programs. This paper presents a framework that enables aggressive program transformations to be applied in the presence of precise exceptions. Our approach relies on the following intuition. First, the program state that needs to be preserved when an exception is thrown is often a very small subset of the program state that is conservatively preserved by compilers to support precise exceptions. By identifying this subset, many spurious constraints on instruction reordering can be removed. Second, exceptions are rarely thrown by correctly executing Java programs,[1] and so it is desirable to optimize the program for this expected case even at the expense of some inefficiency when an exception is thrown.

This paper makes the following contributions:

- It presents both dynamic and static analyses to identify the subset of program state that needs to be preserved if an exception is thrown. In particular, it presents a novel approach to optimize procedures based on runtime propagation of relevant properties of the calling environment to callee procedures. This allows dependences between PEIs and other instructions that modify the value of variables not live at the exception handler to be ignored during instruction-reordering optimizations.
- It presents a framework that allows dependences among PEIs to be completely ignored during program transformations that reorder instructions. This framework is based on a novel algorithm to generate compensation code, which ensures that the same exception is raised as in the original unoptimized program when an exception is thrown. Our algorithm does not rely on any special hardware support and does not incur the overhead of executing *any* additional instructions in the expected case when no exception is thrown. To the best of our knowledge, no other approach has this property.

This paper also presents measurements of relevant benchmark characteristics and preliminary experimental results. A significant percentage of instructions in the benchmarks are PEIs, pointing to the importance of our technique for overcoming dependences among PEIs, which can be applied to any code with PEIs. In 11 out of 13 benchmarks, over 65% of methods are recognized as targets of our aggressive optimization techniques for ignoring program-state dependences

[1] This is particularly true for the predefined *runtime* exceptions, which typically account for most of the exception checks in Java programs. User-defined exceptions are sometimes used for normal control flow.

at PEIs; and using dynamic analysis, over 96% of the method invocations can be aggressively optimized in 9 of those benchmarks. We also demonstrate that significant speedups, up to a factor of 7 on small programs, can be obtained using our techniques and hand-application of well-known program transformations.

The rest of the paper is organized as follows. Section 2 describes background for this work. Section 3 describes the first part of our framework which eliminates spurious dependence constraints by identifying the subset of program state that needs to be preserved if an exception is thrown. Section 4 describes the second part of our framework which allows dependences among PEIs to be completely ignored for performing program optimizations. Section 5 demonstrates empirically that our techniques are applicable to real Java programs and suggests that these techniques can lead to substantial improvement in the performance of programs. Section 6 describes related work and Section 7 presents conclusions.

2 Background

This section reviews background material on the specification of exceptions in Java and on the modeling of exceptions in our compiler's intermediate program representation.

2.1 Exceptions in Java

Java defines `try-catch` blocks that govern control flow when an exception is thrown by a statement – execution proceeds to the closest dynamically enclosing `catch` block that handles the exception [14]. Java exceptions (represented by `Throwable` classes) fall into one of four categories: *runtime exceptions*, *checked exceptions*, *errors*, and *asynchronous exceptions*. Of these, only the first two require a compiler to generate code that precisely preserves the program state when the exception is thrown. Runtime exceptions include several language-defined exceptions, such as `NullPointerException`, `IndexOutOfBoundsException`, and `ArithmeticException`. Several bytecode instructions can potentially throw a runtime exception, such as field-access instructions, array-access instructions, integer-division instructions, and method-call instructions.

If a thread fails to catch a thrown exception, it executes the `uncaughtException` method of its *thread group*, and then terminates [14]. If the `uncaughtException` method is not overridden by the application for the thread group of the excepting thread, eventually the `uncaughtException` method of the `system` thread group is invoked. This default method simply prints the stack trace at the point of exception.[2] Any other threads associated with the application are not directly affected. It is possible for the user to override the exception handler that catches an exception not caught by any handler in the user code, by specifying the `uncaughtException` method for a thread group.

[2] It does not use any other part of the program state and is followed by the termination of the thread, a fact that we exploit, as described in Section 3.

2.2 Basic Framework to Model Exceptions

We use a form of the *program dependence graph* [7] to model control and data dependences, with a special representation for exception-related dependences. This representation exploits ideas from the *factored* control flow graph (FCFG) [9] to model control flow due to exceptions. The FCFG representation does not terminate basic blocks at PEIs, which results in larger basic blocks and fewer edges than a regular control flow graph. The low-level intermediate representation (LIR) of the Jalapeño optimizing compiler [6] we employ for our optimization uses *condition registers* to represent the *exception-conditional dependences.* An instruction that is exception-conditionally dependent on exception-check instructions should execute only if none of these exception-check instructions indicates that an exception should be thrown. (We treat exception-conditional dependences in the same manner as data or control dependences in this paper.)

In addition to the usual control and data dependence edges, our program dependence graph (which we subsequently refer to as the *dependence graph*) has edges for two kinds of precise-exception related dependences: (i) *program-state* dependences, which ensure that a write to a nontemporary variable is not moved before or after a PEI, in order to maintain the correct program state if an exception is thrown, and (ii) *exception-sequence* dependences among PEIs, which ensure that the correct exception is thrown by the code. Our goal is to eliminate as many of these precise-exception related dependences as possible, so that a program can be optimized without paying a heavy performance penalty for the precise exception semantics of Java. In the following, we will refer to these precise-exception related dependences simply as *exception-related dependences.*

Program Example: Fig. 1 shows a simple Java class containing a method and a low-level intermediate representation, LIR, with optimization. Condition registers, such as $c1$ at dependence graph node $P1$, represent the exception-conditional dependences of instructions on exception-check instructions. For example, instructions of nodes $n2$ and $n3$ should be executed only if the `null_check` instruction of $P1$ succeeds without throwing the `NullPointerException`. As shown in the figure, the Jalapeño optimizing compiler removes redundant instructions for exception checking [7].[3] However, the compiler does not move stores across PEIs or reorder PEIs.

Fig. 2 shows the dependence graph (before applying our techniques) for the LIR shown in Fig. 1. The regular data and control dependence edges are shown as thin lines, the program-state dependences are shown as dotted lines, and the exception-sequence dependences are shown as dashed lines. We have not shown the dependences that are transitively satisfied by other sequences of dependences (e.g., the program state dependence from $P7$ to $n9$ is covered by dependences from $P7$ to $n8$ and from $n8$ to $n9$). The critical path of the graph shown in the figure has a length of 10: $P1$, $P5$, $n6$, $P7$, $n8$, $n9$, $P10$, $n11$, $P12$, $n13$, and $n14$.

[3] For example, only one null check is performed on variable `p`, even though it is dereferenced three times in the original program.

```
public class EXCEPT {
   int f, g;
   static void foo(int a[], int b[], int i, EXCEPT p) {
       a[i] = p.f + p.g;
       b[i] = p.g;
   }
}
```

```
Bytecode                                                  Dependence
Offset    Operator    Operands                            Graph Node
------    --------    --------                            ----------
          label       B0
  3   PEI null_check  c1 = p                                 P1
  3       int_load    t0 = @{p, -16}, <EXCEPT.f>, [c1]       n2  // t0 = p.f
          int_load    t1 = @{p, -20}, <EXCEPT.g>, [c1]       n3  // t1 = p.g
 10       int_add     t2 = t0, t1                            n4  // t2 = t0 + t1
 11   PEI null_check  c2 = a                                 P5
          int_load    t5 = @{a, -4}, [c2]                    n6  //
 11   PEI trap_if     c3 = t5, i, <=U, [c2]                  P7  // bounds_check
          int_shl     t6 = i, 2, [c2, c3]                    n8
 11       int_store   t2, @{a, t6}, [c2, c3]                 n9  // a[i] = t2
 18   PEI null_check  c4 = b                                 P10
          int_load    t7 = @{b, -4}, [c4]                    n11 //
 18   PEI trap_if     c5 = t7, i, <=U, [c4]                  P12 // bounds_check
          int_shl     t8 = i, 2, [c4, c5]                    n13
 18       int_store   t1, @{b, t8}, [c4, c5]                 n14 // b[i] = t1
 19       return                                             n15
          end_block   B0
```

Fig. 1. An example Java code segment and its LIR. Instructions introduced by optimization do not have a bytecode offset. Operands beginning with "c" are condition registers.

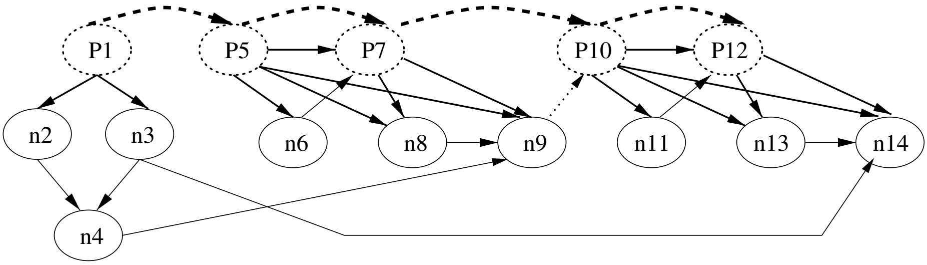


Fig. 2. Dependence Graph for LIR in Fig. 1

.

3 Elimination of Program State Dependences

This section describes an algorithm to identify the subset of program-state dependences that need not be preserved at each PEI. For the purpose of dependence analysis, each PEI, p, represents a use of all variables, i.e., memory locations, that are live (possibly used before defined) on entry to any handler for the exception potentially thrown by p. We refer to this set as $use(p)$. (Section 4 presents a technique to allow exception-sequence dependences to be effectively ignored for the purpose of optimizations.) Section 3.1 describes an analysis that identifies the set of variables that are live on entry to a given exception handler. This analysis takes multithreading into account. Although it is presented separately for clarity, the analysis is applied as part of the algorithm, described in Section 3.2, to propagate information about live variables at enclosing exception handlers to various program points. We describe two algorithms, one using runtime analysis and the other using compile-time analysis, to perform this propagation. Any of these algorithms may be used in a static or a dynamic compiler. Section 3.2 also describes how the liveness information is used to eliminate spurious program state dependences. Section 3.3 describes an extension to improve the quality of the analysis.

3.1 Liveness Analysis at Exception Handler

In this work, liveness analysis is performed at exception handlers in a very coarse-grain manner, so that the information can be compactly represented and propagated along the calling chain of methods in an efficient manner. In particular, we try to capture the common case where only the exception object and some I/O-related variables, such as files and stream objects, are live on entry to an exception handler. We refer to these variables as *exception status-reporting* (ESR) variables. The set of live variables at entry to an exception handler H is

$$L_{IN}(H) = \mathit{UE}(H) \cup [L_{OUT}(H) - \mathit{MustDef}(H)],$$

where $\mathit{UE}(H)$ is the set of variables with *upward-exposed use* (i.e., a use not dominated by a must definition) in H, $L_{OUT}(H)$ is the set of live variables at exit from H, and $\mathit{MustDef}(H)$ is the set of variables definitely written in H. $L_{OUT}(H)$ must include variables that are live not only in the current thread, but also in other threads.

We compute $\mathit{UE}(H)$ by scanning the `catch` block code to check if it only uses ESR variables or local variables declared within the `catch` block (local variables don't contribute to an upward-exposed use for the block). Otherwise, $\mathit{UE}(H)$ is conservatively set to include all nonlocal variables (i.e., variables not locally declared in H) that are visible in H[4] – in this case, further computation of $L_{OUT}(H)$ is not needed. For simplicity, one can conservatively use $\mathit{MustDef}(H) = \emptyset$ in the liveness computation.

[4] Local variables of a method are not visible to an exception handler appearing outside that method.

$L_{OUT}(H)$ is computed as follows:

$$L_{OUT}(H) = LE_{OUT}(H) \cup LNE_{OUT}(H).$$

These two components, $LE_{OUT}(H)$ and $LNE_{OUT}(H)$, respectively account for the following two possibilities: (1) execution of the handler H itself ends abnormally with an exception being thrown (possibly due to a PEI in H), or (2) the execution of the handler ends normally.

We obtain the first component as

$$LE_{OUT}(H) = \bigcup_{e \in E} L_{IN}(EEH(H, e)),$$

where E denotes the set of exceptions possibly thrown by H, and $EEH(H, e)$ denotes the set of enclosing exception handlers that can catch the exception e thrown by H. This component is computed as part of the algorithm, described in Section 3.2, to identify the dynamically enclosing exception handlers for different exceptions at an arbitrary point in the program.

To compute $LNE_{OUT}(H)$ (liveness at exit from H, when no exception is thrown by H), we use a simple algorithm that checks for one of the following cases: (i) H ends with a system exit, (ii) H ends with an exception being thrown by an explicit `throw` statement, (iii) H is followed by termination of the current thread, or (iv) default, when none of the other cases apply. In the first case, where the `catch` block code terminates by calling the `System.exit` method, $LNE_{OUT}(H) = \emptyset$, because the entire application is terminated after H is executed. In the second case, again, $LNE_{OUT}(H) = \emptyset$, because the execution of H does not end normally.[5] This case includes a common idiom where the `catch` block code rethrows the same exception after performing some bookkeeping action, such as releasing a lock. In the third case, which holds for the default `uncaughtException` method of the `System` thread group, $LNE_{OUT}(H)$ is conservatively estimated as the set of variables that are not local to the current thread. *Escape analysis* [10,4,5] can be used to identify thread-local objects. In particular, the algorithm presented in [10] has been shown to often identify a large percentage of objects as thread-local. In the fourth case, $L_{OUT}(H)$ is conservatively set to include all variables visible on exit from H.

Finally, computing the union of $L_{OUT}(H)$ with $UE(H)$, conservatively assuming $MustDef(H) = \emptyset$, leads to one of the following possible values of $L_{IN}(H)$: (i) only ESR variables, (ii) ESR variables and variables that are not thread-local, or (iii) all nonlocal variables (variables not declared in H) visible on exit from H – we shall refer to such an exception handler as *nontrivial.* Two bits are sufficient to encode this liveness information for a given handler. We can use one of the bits (say, bit 0) to encode the liveness of thread-local variables, and the other bit to encode the liveness of variables that are not thread-local (always implicitly assuming that the ESR variables are live). Therefore, the three possible values above are encoded as, respectively, 00, 10, and 11. This encoding has the advantage that a union operation can be performed using a bitwise *or* operation,

[5] The computation of $LE_{OUT}(H)$ completely accounts for $L_{OUT}(H)$ in this case.

which is convenient when a single word is used to record liveness information for multiple handlers for various exception types. It is conceptually straightforward to extend our framework to use more precise (and expensive) liveness analysis.

3.2 Identification of Enclosing Exception Handlers and Propagation of Liveness Information

A simple control flow analysis of `try-catch` blocks, together with static type information, is sufficient to identify the possible enclosing exception handlers, if any, for a PEI within its method. If the type of exception thrown by a PEI can be assigned to (is either the same or a subtype of) the declared exception type of the `catch` clause, we regard the exception handler (corresponding to that `catch` block) as enclosing the PEI. In a `try-catch-finally` construct [14], we regard the `finally` block as enclosing each PEI in the `try` block.

An instruction that does not appear in a `try` block within a method may still be dynamically enclosed in a `try` block due to a calling ancestor of the given method. We describe two interprocedural algorithms to obtain information about the dynamically enclosing exception handler(s) that may catch an exception thrown by an instruction: one is a dynamic analysis, i.e., it uses runtime information, and the other is a static analysis; it uses purely compile-time information.

Consider a call by method A to method B inside a *try* block with k *catch* clauses, where each pair $[H_i, E_i], 1 \leq i \leq k$ represents the exception handler H_i and the exception type E_i caught by it. The basic idea is to propagate the liveness information about the enclosing exception handler(s) (LEEH), $([L_{IN}(H_i), E_i], 1 \leq i \leq k)$, to the callee node B and to each method transitively called by B. Our algorithm using dynamic analysis performs this propagation precisely to the relevant methods, while the one using static analysis may conservatively propagate this information to methods where it is not necessary. In both algorithms, an initial compile-time analysis is performed separately to identify any method that overrides the `uncaughtException` method of any thread group. If such a method H' exists, the pair $[L_{IN}(H'), \mathit{Exception}]$ is conservatively added to the LEEH information for the main method, where *Exception* denotes the Java `Exception` class from which all exceptions that need to be handled precisely are derived.

Dynamic Analysis The key idea behind our dynamic analysis is to propagate runtime liveness information about enclosing exception handlers to each method activation. This information is passed in the form of an extra LEEH parameter. The set of exceptions tracked by this analysis can be determined by a compile-time linear scan of the complete program to identify the set of exceptions potentially thrown by any instruction in the program.[6] For simplicity, in this work, we do not perform such a scan, but instead keep separate liveness information only for the predefined runtime exceptions in Java (i.e., we use two bits,

[6] In general, the complete program is required for such an analysis.

as described in Section 3.2, for each distinct runtime exception and its superclasses). Information on all other handlers, which catch checked, asynchronous, or user-defined exceptions, is summarized with two additional bits, without further distinction among exceptions. Our results, presented in Section 5, suggest that runtime exceptions account for most of the exception checks in typical Java programs.

The total number of predefined runtime exceptions (e.g., the `NullPointerException`) and their superclasses (`RuntimeException` and `Exception`) is small enough that the LEEH parameter can be encoded in a 32-bit word. The LEEH parameter for the `main` method of the application is initialized to all "10" values for each exception type (representing the liveness of only nonthread-local variables and ESR variables at the default `uncaughtException` method provided by the JVM), unioned (using a bit-wise *or* operation) with the liveness encoding for each overriding `uncaughtException` method declared in the program. At each call site not appearing in a `try` block, the LEEH formal parameter value of the caller is passed, without any change, to the callee as the LEEH actual parameter. At a call site inside a `try` block, we require two values to accurately compute the LEEH value passed to the callee: *GEN*, which denotes the encoding of liveness information at the corresponding `catch` blocks (computed as described in Section 3.1), and *KILL*, which is used to override the LEEH information from the caller for exceptions that are definitely caught by one of the given `catch` blocks, and therefore will not be caught by a handler dynamically enclosing the caller. *KILL* is simply encoded as another 32-bit word with the bits "00" appearing for each runtime exception type that is caught by one of the given `catch` blocks, and 1's appearing in all other positions. The LEEH parameter value for the caller is obtained as

$$LEEH_{callee} = (LEEH_{caller} \;\; \& \;\; KILL) \;\; | \;\; GEN,$$

where "&" and "|" represent the bit-wise *and*, *or* operations respectively.

In a dynamic compiler, the runtime information about the actual exception handlers enclosing a method call can be used to optimize the code generation of that method at runtime. Therefore, in a dynamic compiler, no program state dependences are imposed between a PEI and write statements for variables that are not live on entry to the exception handler for that PEI. Alternatively, in a static or dynamic compiler, *cloning/specialization* [21] can be used to obtain different versions of the method, optimized to different degrees based on the incoming LEEH information. At one extreme, only a single additional version of the method may be created to handle the best case where none of the PEIs in the method has an enclosing exception handler with live variables other than ESR variables. We used this approach in obtaining the results presented in Section 5. Furthermore, execution history information about a method may be used to create the specialized version that is most likely to be selected at runtime. We do not discuss these techniques further in this paper.

Consider the example in Fig. 3, where method `main` calls `leaf1` twice, once from within a `try-catch` block and once outside of any such catch block. Because the `catch` block inspects static data, we consider all variables to be live at

```
 public class LIVENESS {
   static int glob;
   static void main() {                     static void leaf1() {
     try {                                    ...
S1:    leaf1();                             }
     }
     catch (NullPointerException e) {       static void leaf2() {
S2:    ... = glob                             ...
     }                                      }
S3:  leaf1();
S4:  leaf2();
   }
 }
```

Fig. 3. Example for static and dynamic liveness analysis

the entry to the handler. However, the second call (at `S3`) does not have such a handler. The dynamic analysis will augment the calls to encode this liveness information at the different calls, allowing for the elimination of spurious program state dependences when `leaf1` is called from `S3`.

Static Analysis The static analysis is performed on the *call graph* representation of the program. In the call graph, each node represents a method, and an edge e_i from node A to node B represents a call site in A that invokes B. For a virtual method call, there is a separate edge for each potential target of the method. Given a call to method B inside a `try` block, the liveness information about enclosing exception handlers (LEEH) is propagated to the callee node(s) for B and to each node reachable from that node along the call graph edges. This ensures, conservatively, that liveness information is propagated to each method with an activation that is dynamically enclosed by the given `try-catch` block. At each method, liveness sets from its different call sites are combined using a union operation.

Finally, for each PEI, the compiler obtains information (using interprocedural analysis described above and using intraprocedural analysis of local `try-catch` blocks) about the live variables at each potentially enclosing exception handler. This information is used to restrict the program state dependences that are imposed between PEIs and write statements in that method.

Our static analysis is conservative relative to the dynamic analysis in two ways. First, for a call to a virtual method inside a `try` block, the enclosing exception handler information is propagated to all methods considered to be targets of that virtual call based on static type analysis. Second, information on call paths is ignored. If a method is called without an enclosing handler (and hence, with the propagation of liveness set of only the default `uncaughtException` handler) along many call paths, but called with a nontrivial handler along even a single call path, all the execution instances of the method are considered to have a nontrivial enclosing handler. On the other hand, the advantage of static analysis is that no extra method parameter is needed to record the runtime liveness information.

Once again consider the example in Fig. 3. Using static analysis, liveness information for all calls is unioned. For example, method `leaf1` in Fig. 3 will be regarded as being enclosed by a catch block of `NullPointerException` due to the call at `S1` although the call at `S3` is not in any catch block. This results in the need to honor program state dependences in `leaf1` (because of the call from `S1`). Program state dependences involving thread-local variables need not be honored in `leaf2` because it does not have a call enclosed by a catch block.

3.3 Impact of Java Memory Model

Our basic analysis, described in Section 3.1, regards all variables visible to other threads as live at an exception handler, unless the exception handler leads to termination of the entire application. Thus, it forces all write statements for those variables to have dependences with respect to the corresponding PEIs. This is a consequence of the current Java memory model not requiring conflicting data accesses in different threads to be synchronized in order to guarantee defined behavior. However, the current memory model has some "serious flaws" in this regard [22] and is in the process of being revised [22,17]. If, as expected, it is revised to require synchronization between writes and reads of a variable in different threads, a write operation performed in an unsynchronized region may be regarded as not necessarily visible to any other thread. (We refer to statements that are dynamically enclosed in a synchronized method or synchronized statement block as constituting a synchronized region). Note that when a PEI p throws an exception within a synchronized region, the default exception handler releases the lock held by the thread, and in that case, we must regard the program state reflecting the effect of instructions up to the excepting instruction as visible to other threads. We identify synchronized regions in our preliminary implementation to demonstrate the additional precision that can be obtained if the revised Java memory model requires synchronization between writes and reads of a variable in different threads.

4 Ignoring Dependences Among PEIs

This section describes how we completely eliminate exception-sequence dependences between PEIs for the purpose of applying program optimizations. The key contribution of this section is an algorithm to generate two sets of code: the *optimized code* that runs ignoring the dependences among PEIs (without relying on any special hardware and without any check-pointing overhead); and the *compensation code* that runs only when an exception occurs during the execution of the optimized code. The role of the compensation code is to intercept any exception thrown by the optimized code, and throw the same exception that would be thrown by the unoptimized code. Section 4.1 describes a transformation that is performed on the dependence graph. Section 4.2 proves the correctness of this transformation. Section 4.3 describes a modified code generation algorithm and proves its correctness.

The dependence graph that is the basis of the transformation described in this section reflects the results of applying the techniques described in Section 3: the dependence graph contains, along with the regular control and data dependences, any remaining program-state dependences for precise exception semantics that are not eliminated by the techniques described in Section 3.

4.1 Transformation of the Dependence Graph

We perform three transformation steps on the unoptimized dependence graph. The first step splits each PEI node, PEI_i, into two separate nodes: (a) an *exception-monitoring node*, em_i, that determines whether an exception should be thrown and sets its *exception flag*, called ef_i, to `true` or `false`, accordingly; and (b) an *exception-throwing node*, et_i, that actually throws an exception if ef_i is set `true` by em_i. After this PEI-node splitting, the original exception-sequence dependences are preserved by the second step, which inserts an exception-sequence edge from em_i to et_i for each PEI_i, and replaces each exception-sequence edge from PEI_i to PEI_j with an exception-sequence edge from et_i to em_j. Finally, the third step replaces each exception-sequence edge from et_i to em_j with an exception-sequence edge from et_i to et_j. This step, therefore, has the effect of eliminating the exception-sequence dependences among exception-monitoring nodes.

The transformed dependence graph consists of two components: the *optimized dependence subgraph*, which does not have any exception-throwing nodes, and the *exception-thrower subgraph*, which has only exception-throwing nodes. The transformed dependence graph allows two additional optimizations not possible with the unoptimized dependence graph: (1) delaying throwing exceptions by et_i after em_i sets ef_i true, and (2) reordering exception-monitoring nodes (modulo control/data dependences). The first comes from splitting em_i and et_i, and the second comes from deleting the exception-sequence edges among em nodes.

Let us revisit the LIR shown in Fig. 1. The *transformed dependence graph* in Fig. 4 is the result of this transformation applied to the unoptimized dependence graph in Fig. 2. Of the two subgraphs of the transformed dependence graph, the optimized subgraph (in Fig. 4-(B)) becomes the basis for our optimization. The critical paths (there are two) of the optimized subgraph alone have lengths of 4 each: $em5$, $n6$, $em7$, $n8$, $n9$; and $em10$, $n11$, $em12$, $n13$, $n14$. Shorter critical paths in the dependence graph result in more freedom in code reordering. (In the figure, the program-state dependence from $n9$ to $em10$ has been deleted after liveness analysis.)

The next section shows that generating optimized code, based on honoring the dependences in the transformed dependence graph and skipping the execution of certain statements when an exception flag is set, leads to no violation of the precise exception semantics. In Section 4.3, we describe a modified algorithm for code generation that leads to more efficient optimized code, and prove its correctness as well.

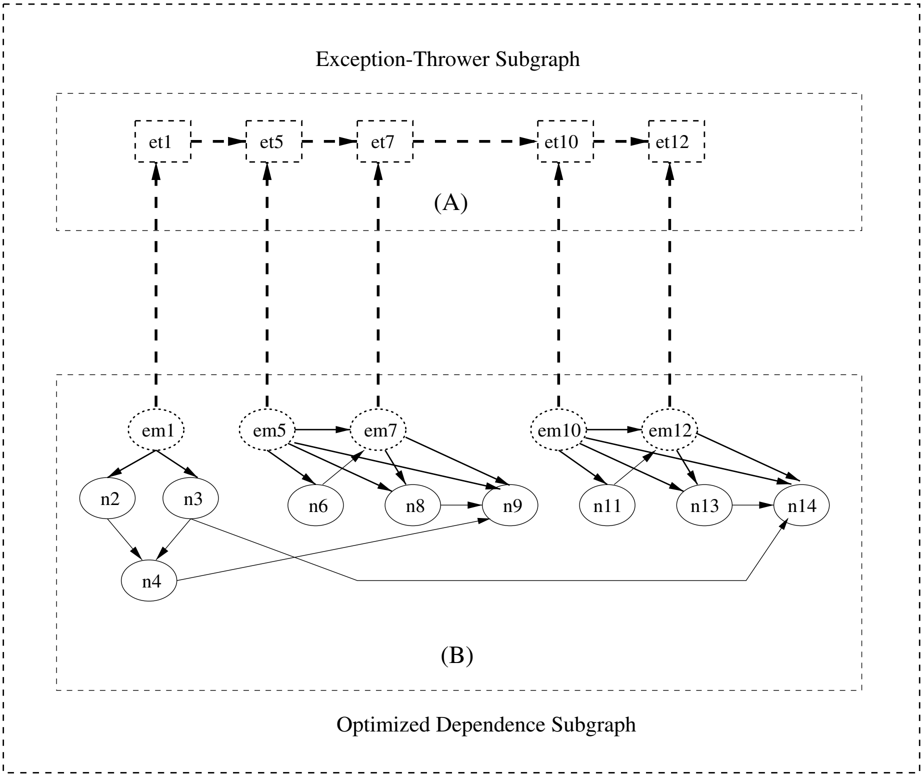


Fig. 4. Transformed dependence graph for LIR in Fig. 1. (A) is the exception-thrower subgraph. (B) is the optimized dependence subgraph. (A) and (B) combined constitute the transformed dependence graph. Rectangular nodes in (A) are exception-thrower nodes, and dotted oval nodes in (B) are exception-monitoring nodes.

4.2 Correctness

This section proves the correctness of the optimized code for the case when the control flow graph (CFG) is acyclic, i.e., with no loops in the procedure. We show the correctness by proving that the code generated from the transformed dependence graph throws the same exception that the code generated from the original dependence graph. Our transformation is valid in the presence of arbitrary control flow, and the proof can indeed be extended to a cyclic CFG by employing the concepts of loop iteration vectors and loop-carried dependences.[7] We omit the more general proof due to its complexity.

We first define a few terminologies to be used below as follows:

- P_o denotes the optimized code, and P_u denotes the unoptimized code.

[7] In general, the control flow graph (CFG) need not be preserved in order to have a program transformation that preserves the program semantics, as long as the control/data dependences are preserved by the transformation.

- $n_i \xrightarrow{*} n_j$ indicates that there exists a CFG path, possibly zero length, from n_i to n_j. In other words, n_i executes before n_j (because we assume an acyclic CFG).
- $n_i \xrightarrow{es} n_j$ indicates that there exists an exception-sequence dependence edge from n_i to n_j.
- $n_i \xrightarrow{es,*} n_j$ indicates that there exists an *exception-sequence path* (a path over exception-sequence dependence edges), possibly zero length, from n_i to n_j.
- $\mathit{Slice}(n_i)$ denotes the set of nodes in the transformed dependence graph that are reachable from n_i. In other words, $n_j \in \mathit{Slice}(n_i)$ if and only if there exists a path from n_i to n_j in the transformed dependence graph.
- $\mathit{Slice}_{ec}(n_i)$, a subset of $\mathit{Slice}(n_i)$, denotes the set of nodes in the transformed dependence graph that are reachable from n_i over *non*-exception-sequence (dependence) edges. In other words, $n_j \in \mathit{Slice}_{ec}(n_i)$ if and only if there exists a path over control/data dependence edges, possibly zero length, from n_i to n_j in the transformed dependence graph. For example, $\mathit{Slice}_{ec}(em1) = \{n2, n3, n4, n9, n14\}$. None of the exception-thrower nodes belongs to $\mathit{Slice}_{ec}(x)$, for any x, because they are reachable only via exception-sequence edges.

For convenience, PEIs are indexed from 1 to N based on a topological ordering of the PEI nodes: if $i < j$, there is no execution path from n_j to n_i (because we assume an acyclic CFG). We also use min_{EX} to denote the minimum number among the indices of the exception-flags that are `true`, i.e.,

$$min_{EX} = k \text{ if } \forall j, 1 \le j < k, ef_j = \texttt{false} \wedge ef_k = \texttt{true}.$$

Note that if $min_{EX} = k$ in P_u, PEI_k is the (first) one that throws the exception in P_u.

Our optimization scheme ensures that the correct exception is thrown by P_o when multiple *em* nodes in P_o set their exception flags (to `true`) by skipping the execution of $\mathit{Slice}_{ec}(em_j)$ in P_o if $ef_j = \texttt{true}$. The following theorem shows the correctness of this scheme.

Theorem 1. *Let the execution of $\mathit{Slice}_{ec}(em_j)$ in P_o be skipped if $ef_j = \texttt{true}$. Then, $min_{EX} = k$ in $P_o \iff min_{EX} = k$ in P_u.*

Before proving the theorem, we first present Property 1, which states the condition for a correct out-of-order execution of instructions in an optimized code. We assume that any out-of-order optimization performed by the compiler observes this property.

Property 1. Let $n_i \xrightarrow{*} n_j$ in P_u. Execution of n_j can precede that of n_i in P_o only if $n_j \notin \mathit{Slice}(n_i)$.

We prove Theorem 1 based on the following lemmas.

Let $EM = em_{m(1)} \xrightarrow{*} em_{m(2)} \cdots \xrightarrow{*} em_{m(k)}$ in P_o be an out-of-order execution of exception-monitors such that $m(1) > m(2) > \cdots > m(k)$ in P_u.

Lemma 1. *Let the execution of $Slice_{ec}(em_{m(i)} \in EM), 1 \leq i < k$, in P_o be skipped. Then,*

$$ef_{m(k)} = \texttt{true} \textit{ in } P_o \Longleftrightarrow ef_{m(k)} = \texttt{true} \textit{ in } P_u.$$

Proof: Let j be any $m(i), 1 \leq i < k$. The only side effect of $em_j \in EM$ is setting ef_j. Therefore, executing em_j before $em_{m(k)}$ does not affect the program state of $em_{m(k)}$. Also, $PEI_{m(k)} \xrightarrow{*} PEI_j$ means there exists an exception-sequence path from $em_{m(k)}$ to et_j, which prevents et_j from prematurely throwing an exception before $em_{m(k)}$ executes. Finally, any node $n_q \in Slice(em_j)$ comes after $em_{m(k)}$ in P_u because $PEI_{m(k)} \xrightarrow{*} PEI_j$, and skipping them should not affect the program state of $em_{m(k)}$. □

Lemma 2. *Let $ef_{m(i)} = \texttt{false}, 1 \leq i < k$. Then,*

$$ef_{m(k)} = \texttt{true} \textit{ in } P_o \Longleftrightarrow ef_{m(k)} = \texttt{true} \textit{ in } P_u.$$

Proof: Again, let j be any $m(i), 1 \leq i < k$. Any node $n_q \in Slice_{ec}(em_j) \cup em_j$, that is executed before $em_{m(k)}$ must not affect $em_{m(k)}$ for correct optimization, as stated by property 1. Since $ef_j = \texttt{false}$, executing $n_q \in Slice_{ec}(em_j)$ will not result in any illegal operation, either. □

Proof (of Theorem 1): According to Lemma 1 and Lemma 2, either skipping the execution of $Slice_{ec}(em_{m(i)} \in EM)$ or $ef_{m(i)} = \texttt{false}, 1 \leq i < k$, ensures that

$$min_{EX} = m(k) \text{ in } P_o \Longleftrightarrow min_{EX} = m(k) \text{ in } P_u.$$

Therefore, skipping the execution of $Slice_{ec}(em_{m(i)})$ when $ef_{m(i)} = \texttt{true}$ still ensures the same. □

Theorem 1 also shows how optimized code might be generated to guarantee the same exception to be thrown that an unoptimized code will throw: when ef_i is `true` all the statements that are transitively control/data dependent on em_i should not be executed (in order not to cause any illegal operations). This scheme, however, penalizes an exception-free execution because execution of statements need to be guarded by code that checks whether the corresponding exception flag is `true` or `false`. The code-generation scheme we employ eliminates this unwanted penalty for exception-free execution at an increased cost of *exceptional execution*, i.e., when an exception is thrown. This shifting of additional cost from an exception-free execution to exceptional execution, we believe, is the right approach for optimization because exceptions are in general regarded as being *exceptional.* The following section describes our code-generation scheme in detail.

4.3 Generation of Optimized Code and Compensation Code

The core idea of the code generation algorithm is generating two sets of code: the *optimized code* that runs ignoring the dependences among PEIs; and the *compensation code* that runs only when an exception occurs during the execution of the optimized code. The optimized code is generated from the optimized dependence subgraph, such as the one shown in Fig. 4-(B). The compensation code is generated from the fully transformed dependence graph, such as the one shown in Figs. 4-(A) and 4-(B) combined, and its behavior follows Theorem 1. The role of the compensation code is to intercept any exceptions thrown by the optimized code, and to throw the same exception that would be thrown by the unoptimized code. The compensation code does not attempt to recover any "lost" program state: the dependence graph prevents code generation that might require any such recovery of the program state when an exception occurs.

Our strategy is to run the optimized code until an exception is thrown. This exception might not be the same (correct) as the exception thrown by the unoptimized code, but will be caught by the compensation code. Then, the compensation code runs as described in the previous section until an exception-throwing node throws the real exception, which will be the correct exception to be thrown by the unoptimized code. This way, performance of the optimized code does not suffer when no exception is thrown by the optimized code. Transfer of execution from the optimized code to the compensation code when an exception occurs is performed by an exception handler: we enclose the optimized code in a `try` block, whose `catch` block contains the compensation code. In a dynamic compiler, the compensation code can be generated on demand when an exception is thrown. In a static compiler, the compensation code may be placed such that it does not pollute the instruction cache during normal execution when no exception is thrown. The following steps describe the details of code generation:

1. We generate code from the transformed dependence graph. This code is the basis of the compensation code.
2. A copy is created of the compensation code, excluding the portion corresponding to the exception-thrower graph. This code, equivalent to code generated from the optimized dependence subgraph, is the basis of the optimized code. Both the optimized code and the compensation code share the same program state, reading from and writing to the same variables.
3. In the optimized code, we replace each exception-monitor code with the corresponding PEI code.
4. Exception-monitors in the compensation code *only* set the exception flags if the results of their exception checks indicate that an exception should be thrown. Guards are introduced in the compensation code so that if the exception flag of an exception-monitor is set, execution of all the statements transitively control/data dependent on the exception-monitor is skipped.
5. The compensation code is put into a catch block that catches any exception thrown by the optimized code. The catch block of the optimized code starts with code, called the *handler prologue*, that examines the exception thrown by the optimized code. The handler prologue sets the exception flag corresponding to the PEI that threw the exception, and transfers the control flow to the instruction immediately after the exception-monitoring node in the compensation code – additional labels are introduced in the compensation code for this purpose.

The following theorem states that the code generated as described above guarantees throwing the same exception that an unoptimized code will throw.

Theorem 2. *The compensation code executes if and only if an exception is thrown by the unoptimized code. Furthermore, the exception thrown by the compensation code is the same as that thrown by the unoptimized code.*

Proof: We prove the theorem by showing that the execution behavior of the optimized code, the compensation code, the handler prologue, and the `try-catch` structure inserted by the code generator, all combined, is the same as that of the compensation code, denoted as P_c, alone. Theorem 1 already showed that P_c and P_u are equivalent in execution behavior.

Let PEI_t be the PEI that throws the exception in P_o. The execution behavior of P_o up to PEI_t is the same as that of P_c up to em_t because (1) they are identical by construction (of the code generation as described above) except for the replacement of PEI nodes in P_o with em nodes in P_c, and (2) no ef_k in P_c, such that $em_k \xrightarrow{*} em_t$, would have been set (otherwise, PEI_t would not be the first PEI to throw an exception). At PEI_t, the exception handler catches the exception thrown by PEI_t and passes the control flow to the handler prologue, which in turn passes the control flow to right after em_t in P_c after setting ef_t to `true`. After that, the execution continues in P_c, which is already shown to be equivalent to P_u in execution behavior by Theorem 1. □

Fig. 5 shows a high-level view of the generated code for method `foo` in Fig. 1. Let us consider an example situation where the original program throws an exception at P1, but during the execution of BODY_A (the optimized code), P5 throws the first exception, which is caught by the generated catch block. The handler prologue sets flag_p5 to `true`, and branches to LABEL_5 in the compensation code, which appears immediately after the instruction corresponding to P5 in the optimized code. Since the execution behaviors of the optimized code and the compensation code are identical (except for the PEIs), the execution continues with the compensation code as it would with the optimized code. At LABEL_5, since the flag is set for the PEI, the compensation code skips the statements in the program that are transitively control/data dependent on $em5$ (i.e., in $Slice_{ec}(em5)$), and continues its execution (eventually) to LABEL_1. Exception-thrower $et5$ can execute only after Exception-thrower $et1$ because of the exception-sequence dependence from $et1$ to $et5$. Exception-thrower node $et1$ can, in turn, execute only after $em1$, due to the exception-sequence dependence from $em1$ to $et1$ in the exception dependence graph. Therefore, even though P5 initially throws an exception during execution of the optimized code, $et1$ will throw the "real" exception to be caught by the user application or by the Java virtual machine.

While our example uses straight-line code for simplicity, our algorithm for compensation code generation works correctly in the presence of arbitrary control

```
foo() {
    try {
      BODY_A; // optimized code of foo from the optimized graph
    } catch (... e) {

      // handler prologue
      all flags = false;
      if (e was thrown by PEI1) {
         flag_p1 = true;   branch to LABEL_1;
      } else if (e was thrown by PEI5) {
         flag_p5 = true;   branch to LABEL_5;
      } else if (e was thrown by PEI7) {
         flag_p7 = true;   branch to LABEL_7;
      }

      // Compensation code from (B) + (C).
      // Note that code for PEI5 has been code-motioned before PEI1.
      . . .
      LABEL_5: // code immediately after PEI5 monitor in optimized code
      . . .
      LABEL_1: // code immediately after PEI1 monitor in optimized code
      . . .
      if (flag_p1) throw exception(PEI1);
      . . .
      if (flag_p5) throw exception(PEI5);
      . . .

    }
}
```

Fig. 5. A high-level view of the generated code for method `foo` in Fig. 1

flow, including loops. Furthermore, the technique described above can be applied to any region of the method, including a `try` block.

5 Empirical Measurements

This section describes empirical evidence of the effectiveness of some of the techniques described in the paper. It includes (1) evidence that PEIs occur often in practice, particularly those related to runtime exceptions; (2) measurements of the effectiveness of static and dynamic analyses for detecting methods with desirable exception handler properties;[8] and (3) experimental results that demonstrate the potential for performance improvement from our techniques to

[8] We report the effectiveness of our algorithm for overcoming program-state dependences. Our techniques for overcoming the exception-sequence dependences among PEIs are *always* applicable to any code with PEIs.

eliminate exception-related dependences. The static analysis (Section 6) is implemented in an extended version of Jax [25], an application extractor for Java. Liveness information for exception handlers is currently implemented as a conservative version of the analysis described in Section 3.1.

The dynamic analysis is implemented in the Jalapeño Java virtual machine [2] by instrumenting each method using the Jalapeño optimizing compiler [6] to dynamically propagate exception handler information as described in Section 3.2. This implementation does not currently perform liveness analysis of exception handlers, and treats all exception handlers, except for the default `uncaughtException` method, as nontrivial. Neither analysis performs the "kill" computation described in Section 3.2.

Because the Jalapeño optimizing compiler does not yet contain sophisticated code transformations, performance improvements from eliminating exception dependences allowed by our techniques are not yet realized in the system. To illustrate that this is not a shortcoming of our techniques, we hand-optimized two applications at the Java source level, applying transformations that are enabled because of the dependences eliminated by our techniques. The transformations we applied, loop tiling and outer loop unrolling (also known as loop unroll-and-jam) [26,23], are well-known compiler transformations that are expected to be included in the Jalapeño optimizing compiler in the future. Both the original and optimized versions of the applications were executed using Jalapeño.

Table 1 presents some characteristics of the benchmark suite, which includes 13 programs from the SPEC JVM98 suite and other sources. The second column of Table 1 presents the number of statically reachable methods in the original application (including libraries) as determined by Jax's static type analysis. The next column reports the number of HIR instructions for those reachable methods. The next column report the percentage of HIR instructions that can throw a predefined runtime exception, such as `ArithmeticException`, `NullPointerException`, `IndexOutOfBoundsException`. The last column reports this percentage for all PEIs. These results suggest that PEIs are quite frequent — on an average 42.5% of HIR instructions are PEIs, which confirms the importance of overcoming exception-sequence dependences among PEIs. Furthermore, a large percentage of PEIs (on average 99.5%) throw only runtime exceptions predefined by the Java language.[9]

Table 2 presents the effectiveness of the static and dynamic analyses for the techniques described in Section 3. The first two columns after the benchmark name report the results of the static analysis. The next three columns report the results of the dynamic analysis. The first static analysis column reports the percentage of methods that (1) neither contain an exception handler nor have an enclosing exception handler (EEH) in the program, or (2) all such handlers are *trivial*, in that they do not contribute to the liveness set. The next column reports the same metric as the previous column, except it checks only for the

[9] None of benchmarks override the default implementation of `uncaughtException` mentioned in Section 2.

Table 1. Benchmark characteristics sorted by number of HIR instructions. HIR, the high-level intermediate representation of the Jalapeño optimizing compiler, is converted (with optimization) to LIR, which is closer to machine code.[10]

		HIR	Pct of PEIs	
Benchmark	Methods	Instructions	Runtime Only	Total
Cholesky Factorization	24	330	31.2%	32.1%
Matrix Multiply	21	381	35.2%	36.0%
201.compress	140	2,654	37.3%	38.4%
209.db	157	3,069	44.5%	45.7%
227.mtrt	212	4,621	51.6%	51.9%
toba	167	6,422	55.2%	55.6%
pBOB 1.1	190	6,682	45.0%	45.7%
JavaLex	201	8,752	41.4%	41.7%
222.mpegaudio	341	10,316	42.8%	43.2%
228.jack	415	10,868	50.6%	51.3%
202.jess	586	11,228	50.5%	51.4%
JavaParser	654	12,242	20.3%	20.8%
213.javac	1,155	23,742	44.9%	45.4%
Average			42.3%	42.5%

Table 2. Applicability results, "RT" – Runtime, "EEH" – Enclosing Exception Handler. Our instrumentation failed to produce dynamic measurements for `228.jack`.

	Static Analysis		Dynamic Analysis		
	% Methods with		% of Executed Methods with		
	No EEH	No RT EEH	No EEH	No RT EEH	No RT EEH and not Sync Region
Cholesky Factorization	100.0%	100.0%	100.0%	100.0%	100.0%
Matrix Multiply	100.0%	100.0%	100.0%	100.0%	100.0%
201.compress	52.9%	65.7%	99.996%	100.0%	99.998%
209.db	29.9%	69.4%	2.428%	99.968%	72.847%
227.mtrt	67.0%	93.4%	49.218%	96.024%	96.020%
toba	24.6%	88.0%	3.005%	99.995%	96.735%
pBOB 1.1	76.3%	99.5%	6.479%	10.957%	7.933%
JavaLex	1.2%	100.0%	1.238%	100.0%	96.687%
222.mpegaudio	21.1%	25.2%	0.012%	0.034%	0.023%
228.jack	12.3%	80.5%	—	—	—
202.jess	12.3%	69.6%	99.243%	99.996%	99.768%
JavaParser	4.6%	99.8%	0.000%	100.0%	100.0%
213.javac	5.1%	6.3%	0.037%	0.064%	0.042%
Average	39.0%	76.7%	30.135%	67.253%	64.171%

presence of nontrivial *runtime* exception handlers, thereby potentially increasing the percentage of methods.

[10] The number of HIR instructions of a program is slightly smaller than that of LIR instructions of the program. For example, the total numbers of HIR and LIR in-

The static results show that, except for a few benchmarks, the majority of methods either contain an exception handler or are enclosed in a nontrivial exception handler (Static No EEH column). However, as shown by the next (Static No RT EEH) column, in 11 of 13 benchmarks, over 65% of methods neither contain, nor are enclosed in, a nontrivial *runtime* exception handler.[11]

The metrics used for dynamic analysis are the same as those for static analysis, except that the analysis is performed at runtime for each method *activation*. In 9 of 12 benchmarks, over 96% of method activations are not enclosed in a runtime exception handler inside the program, and do not themselves contain a runtime exception handler. Hence, these methods can be optimized quite aggressively, particularly with further application of techniques presented in Section 4. The decrease in effectiveness for the `pBOB` benchmark is mostly due to the conservative liveness analysis used in our current implementation, given the manner in which Java bytecode generators implement synchronized statements. Namely, the compiler translates synchronized statements into a `try-catch` block, where the `catch` block unlocks the locked object and rethrows the exception. Although the static analysis implementation determines that such blocks are trivial, the current dynamic analysis implementation does not have this feature.

The extension described in Section 3.3, as shown in the last column under dynamic analysis, finds a very high percentage of the "optimizable" methods (those without enclosing or local runtime exception handlers, from the second column) as being unsynchronized themselves and appearing in unsynchronized regions. For example, the percentage of methods in `JavaLex` that are not enclosed by a runtime exception handler decreased only marginally from 100.0% to 96.687% when methods appearing in synchronized regions are excluded. With the expected extensions to the Java memory model [22,17] (or by simply not honoring those features of the current model that apply to programs with data races, as all commercial JVMs seem to do [22,17]),[12] these methods could be optimized completely ignoring *all* runtime exception-related dependences.

Table 3 provides an indication of how the techniques presented in this work can affect execution performance for `Matrix Multiply` and `Cholesky Factorization`. It reports the execution time in seconds for the original application and a hand-optimized version that applied well-known transformations, such as loop tiling and outer loop unrolling [26,23]. These transformations are enabled because of the dependences eliminated by our techniques. The results show speedups of 6.88 and 3.34 for `Matrix Multiply` and `Cholesky Factorization`, respectively. Although such speedups may not be realized on larger applications, these results do suggest that promising performance gains can be obtained once

structions of toba are about 23K and 30K, respectively. We have found no significant difference in the number of PEIs between HIR and LIR.

[11] The static and dynamic analysis of the SPEC JVM98 programs begins at the application's main, not the common harness. The harness contains a catch block that catches the `Throwable` exception to allow for robust functionality.

[12] All JVMs tested by Pugh [22,17] were found to break the Java memory model.

Table 3. Performance improvement by eliminating exception dependences and performing hand-transformations

Benchmark	Unoptimized Time (sec)	Optimized Time (sec)	Speedup
Matrix Multiply	115.5	16.8	6.88
Cholesky Factorization	58.8	17.6	3.34

exception-related dependences are eliminated using the techniques described in this work.

6 Related Work

Previous work on speculative code motion for superscalar and VLIW processors [8,24,19,13] has some similarities with our work, in that it involves aggressive code motion and recovery from exceptions thrown by speculative instructions. Broadly, our work differs in at least two ways. First, it does not require any hardware support, while these approaches rely on special hardware to support silent exceptions or to store the results of speculative instructions. Second, (since we are looking at a different problem, that of handling precise exceptions in Java) our work is unique in addressing the problem of reducing the program state that must be preserved at a possible exception point.

The problem of debugging optimized programs also has some similarities with optimizing programs in the presence of exceptions, in that the notion of preserving the *currency* of variables at a breakpoint [15] is similar to that of preserving program state at exception points. Holzle et al. [16] restrict optimizations across *interrupt points* and use dynamic deoptimization of code to support debugging. Other researchers have proposed various approaches to detect variables whose value may not be current at a breakpoint due to program optimizations [11,1,12], and possibly recovering the original value of variables that are not current [12]. In contrast, our work must ensure that the value of each relevant variable is "current" at an exception point and focus on determining the minimal set of such relevant variables. Furthermore, we deal with the problem of ensuring the correct ordering of exceptions (without adversely affecting program optimizations), which does not occur in the context of debugging optimized programs.

Hennessy [15] describes program optimizations in the presence of exception handling, where he essentially deals with the altered control flow due to an exception being thrown. His work does not deal with precise exceptions.

Le [18] describes a runtime binary translator that supports reordering of instructions for architectures that support precise exception semantics. However, this work requires the generation of checkpointing code, which contributes to the overhead of executing extra instructions even when no exception is thrown. Therefore, the benefits of increased flexibility in instruction scheduling have to be weighed against the overhead of checkpointing, which can potentially be high.

Moreira et al. [20] describe, for numeric computations with simple array subscripts, a program transformation to create *safe* regions in which no (out-of-bounds array index or null-pointer) exceptions may take place. They show that the loop-reordering program transformations can contribute up to a factor of 18 improvement in performance on an IBM POWER2 workstation, over and above the improvement from eliminating the exception checks in safe regions. Our work handles the more commonly occurring case where it is not possible to create exception-free regions, for example, if there is memory being allocated for a new variable in the region.

Arnold et al. [3] describe the impact of Java runtime exceptions on a VLIW and single-issue architecture using multiple instruction scheduling techniques. Their simulation results show that exception overhead can be substantially reduced using advanced scheduling techniques, most notably on the VLIW architecture. The smallest overhead was achieved when using known hardware mechanisms for increased speculation. Their simulation results confirm the adverse impact of precise exceptions on program performance, particularly on non-VLIW machines, thus supporting the need for work like ours to enable aggressive optimizations in the presence of precise exceptions.

7 Conclusions

Precise exception support imposes constraints on optimizations such as instruction reordering. The novel framework presented in this paper enables the relaxation of these constraints. By identifying the subset of program state that needs to be preserved if an exception is thrown, the framework enables the removal of many spurious dependences between writes and potentially excepting instructions. The static and dynamic analysis algorithms we presented can be used separately or together to improve the effectiveness of the analysis. Our framework further allows aggressive optimization of the program, ignoring all dependences between potentially excepting instructions, by generating compensation code that is executed only when an exception is thrown. This code ensures that the same exception is raised as in the original unoptimized code. The algorithms are applicable to any language that constrains program transformations due to dependences involving exceptions. The preliminary implementation using a conservative version of our algorithm shows promising results: using the static analysis, in 11 out of 13 benchmarks, over 65% of methods are recognized as targets of our aggressive optimization techniques for ignoring program state dependences at potentially excepting instructions; using the dynamic analysis, over 96% of the method invocations can be aggressively optimized in 9 of those benchmarks. We have also demonstrated that significant speedups, up to a factor of 7 on small programs, can be obtained using our techniques and well-known transformations.

We expect the importance of techniques presented in this paper to grow further, as Java is used more heavily in application areas that require high performance, and also as Java compilers become more mature, and run into the limitations imposed by precise exception semantics while applying aggressive optimizations that involve code reordering.

Acknowledgments. We thank Pat Gallop, David Grove, and Marc Snir for useful technical discussions. We thank Dave Streeter, Frank Tip, and Doug Lorch for their help with the Jax infrastructure and Igor Pechtchanski and Chandra Krintz for help with the code generation used for the dynamic analysis instrumentation. We also thank David Grove, Harini Srinivasan, Frank Tip, and Laureen Treacy for their useful feedback on earlier drafts of this work and Vivek Sarkar for his support of this work.

References

1. A.-R. A.-Tabatabai and T. Gross. Source-level debugging of scalar optimized code. In *SIGPLAN '96 Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.
2. B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.
3. M. Arnold, M. Hsiao, U. Kremer, and B. Ryder. Instruction scheduling in the presence of Java's runtime exceptions. In *12th International Workshop on Languages and Compilers for Parallel Computing*, August 1999.
4. B. Blanchet. Escape analysis for object oriented languages: Application to Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.
5. J. Bodga and U. Hölzle. Removing unnecessay synchronization in Java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, Colorado, November 1999.
6. M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.
7. C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano, and H. Srinivasan. Dependence analysis for Java. In *12th International Workshop on Languages and Compilers for Parallel Computing*, August 1999.
8. P. Chang, S. Mahlke, W. Chen, N. Warter, and W.-M. Hwu. IMPACT: An architectural framework for multiple-instruction-issue processors. In *Proc. 18th International Symposium on Computer Architecture*, pages 266–275, 1991.
9. J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, September 1999.
10. J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1999.
11. M. Cooperman. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems*, 16(3):387–427, 1994.

12. D. Dhamdhere and K. Sankaranarayanan. Dynamic currency determination in optimized programs. *ACM Transactions on Programming Languages and Systems*, 20(6), November 1998.
13. K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proc. 24th International Symposium on Computer Architecture*, pages 26–37, June 1997.
14. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.
15. J. Hennessy. Program optimization and exception handling. In *8th Annual ACM Symposium on the Principles of Programming Languages*, pages 200–206, 1981.
16. U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.
17. Java memory model mailing list. Archive at http://www.cs.umd.edu/p̃ugh/java/memoryModel/archive/.
18. B. C. Le. An out-of-order execution technique for runtime binary translators. In *Proc. 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–158, October 1998.
19. S. Mahlke, W. Chen, R. Bringmann, R. Hank, W.-M. Hwu, B. Rau, and M. Schlansker. Sentinel scheduling: A model for compiler-controlled speculative execution. *ACM Transactions on Computer Systems*, 11(4):376–408, November 1993.
20. J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *ACM Transactions on Programming Languages and Systems*, 1999. (to appear); Also available as IBM Research Report RC 21166.
21. S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.
22. W. Pugh. Fixing the Java memory model. In *ACM 1999 Java Grande Conference*, pages 89–98, June 1999.
23. V. Sarkar. Automatic selection of high-order transformations in the IBM XL Fortran compilers. *IBM Journal of Research and Development*, 41(3), May 1997.
24. M.D. Smith, M.S. Lam, and M.A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proc. 17th International Symposium on Computer Architecture*, pages 344–354, May 1990.
25. F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, November 1999.
26. M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

HERCULE: Non-invasively Tracking Java^{TM} Component-Based Application Activity

Karen Renaud

Department of Computing Science, University of Glasgow,
17 Lilybank Gardens, Glasgow, G12 8RZ, UK.
karen@dcs.gla.ac.uk

Abstract. This paper presents HERCULE, an approach to non-invasively tracking end-user application activity in a distributed, component-based system. Such tracking can support the visualisation of user and application activity, system auditing, monitoring of system performance and the provision of feedback. A framework is provided that allows the insertion of proxies, dynamically and transparently, into a component-based system. Proxies are inserted in between the user and the graphical user-interface and between the client application and the rest of the distributed, component-based system. The paper describes: how the code for the proxies is generated by mining component documentation; how they are inserted without affecting pre-existing code; and how information produced by the proxies can be used to model application activity. The viability of this approach is demonstrated by means of a prototype implementation.

1 Introduction

The motivations for tracking application activity are many and varied. Motivations could be a need to:

- understand the application execution process, especially if the application is distributed or runs on parallel processors;
- provide information needed to carry out system tuning;
- satisfy security requirements;
- provide an audit trail; or
- provide extra support for end-users.

It is the latter aspect which provides the motivation for the research outlined in this paper. The fact that feedback is required by the end-user of an application is no longer disputed [29]. However, the manner of providing this feedback, and standards for ensuring the quality thereof, are still open to debate. Feedback must, at present, be provided for during the development of an application, and it is extremely difficult to remedy applications which provide inadequate feedback, once they are in use. I propose to augment the application's end-user feedback provision by means of external application tracking.

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp. 447–471, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Providing feedback independently of the application can only be done if we have a means for observing application activity. There are various ways to achieve this, and the approach chosen here is to engage proxies which will intercept communications and generate reports containing details about application activity. These reports are relayed to a framework, called HERCULE[1], which uses the information to provide the desired level of feedback.

Since portraying information about application activity in order to augment application feedback is a novel use of the information derived from application tracking, this paper also addresses the visualisation of the information thus obtained, and shows how the visualisation has been used to provide feedback tailored to the needs of users operating in different roles.

The following section will discuss current research into application tracking, feedback and separation of concerns. Section 1.2 will outline the proposed solution to the problem.

1.1 Related Work

This section discusses current research into application tracking, separation of concerns, and feedback.

Application Tracking. Applications can be tracked from two different perspectives, either tracking the user interaction with the application, or watching application interaction with the system. One example of user interface tracking is seen in the work of Trafton and Brock [39] whose system provides a layer between the user interface and the application to keep track of the user's actions, comparing them to an internal representation of various task models, to try to identify the task being done by the user. When a correspondence can be pinned down, the user is offered the option of the sequence being completed automatically. Fawcett and Provost [16] worked on finding ways to predict whether the user of a given account is not the authorised user. They profile each user by characterising their behaviour based on histories of previous sessions. Many researchers have studied the processes and patterns of user interaction with computer systems [5, 9, 22, 25], while Lin *et al* [24] have developed methods for visualising the masses of data collected about user search patterns in a variety of graphical formats, allowing human pattern recognition capabilities to be applied. Chalmers *et al* [12] have developed a methodology to build up Web usage histories for users in a particular community. The user search path is compared to paths of other users within the community and if a match is found, sites visited by the other users will be suggested as being of probable interest.

Other researchers have looked at tracking application use of system resources. Burton and Kelly [7, 8] have developed a tool which traces system calls and provides the ability to re-execute these calls to allow system tuning. Ball and Larus [4] have described algorithms for placing code within programs in order to record program behaviour and performance. Jeffery *et al* [18] introduce the

[1] Named after Hercule Poirot, Agatha Christie's legendary detective.

Alamo monitor program execution monitoring architecture which assists developers in bug-detection, profiling programs and visualisations. Siegle and Hofmann [35] have developed the SUPRENUM microprocessor which uses a hybrid combination of software and hardware monitoring to determine parallel program behaviour. This assists programmers in gaining insight to the execution of their parallel programs. Wybranietz and Haban [41] also use a hybrid approach to observe system behaviour, measure performance, and record system information. They make use of a special measurement processor which runs monitoring software for each distributed node in the system. The information thus derived is displayed graphically and used to improve understanding about run-time system behaviour. Joyce *et al* [19] monitor distributed systems by means of a distributed programming environment called Jade, which assists the programmer in debugging, testing and evaluating distributed systems.

To summarise, tracking can be done either invasively or non-invasively. Invasive tracking requires that changes be made to the application in order to support the tracking. This is risky, since it could be expensive in terms of time and effort to remove the reporting mechanism when there is no longer a need for it. It is also, by definition, application specific, and tracking must be added to each application type individually. Non-invasive tracking does not require an application to be changed in any way, is easily deactivated, and can seamlessly track a variety of applications, but is much harder to accomplish.

Why Feedback? Users, especially novice users of an application, often have no idea of how to use the application, and need to spend time and effort building up an internal model of how the application works. They seldom read manuals, wanting rather to find out for themselves how the system works [11], and they tend to be impatient to get on with their task [6], not wanting to spend hours being taught how to use a system.

Norman [29] argues that in any complex environment — like a new application — one should always expect the unexpected. To deal with the unexpected, Norman concludes that continuous and informative feedback is essential. Learning how a system works is made safer and less risky if the relevant information is easy to find [11]. Chan *et al* [13] have shown that an active feedback system greatly improves user performance. Therefore, *feedback* is far superior to user manuals for helping the user to build up a correct internal model. The role of clear explanations in this process is vital [23]. Users also need feedback because they have severe limits with respect to the following, taken from [31]:

- *perception* — perceiving small difference in detailed information;
- *memory* — the limitations of human memory is illustrated by the following examples:
  - users sometimes forget what they have done, especially if they are interrupted during a processing session;
  - users often do not detect their errors. Often the user is vaguely aware that something has gone wrong, but has no idea how this occurred.
  - difficulties are often experienced in holding recently experienced information until needed; and

  - users experience problems retaining information retrieved from long-term memory — such as remembering where they are in a plan of action.

On the other hand, users have particular strengths with respect to [31]:

- processing visual information rapidly, and coordinating multiple sources of information;
- making inferences about concepts or rules from past experiences;
- storing common patterns, like user action sequences, efficiently;
- retrieving relevant information quickly.

It is clear that the user requires feedback, to help them to discover how an application works, to help discover what effect their inputs have on the application, and to serve as a memory aid. The feedback is traditionally provided from within the application code, but this approach is flawed because programmers are seldom trained to provide adequate feedback, and it is almost impossible to augment the feedback once the application has been delivered. Users functioning in different roles often have completely different feedback needs, and it is difficult for an application to provide for all of them adequately.

Separation of Concerns. Programmers have to deal with a great deal of complexity as part of their task in developing software. They have to concentrate not only on the required functionality of the software, but also with important issues like replication, distribution, real-time configuration, synchronisation, perisistence and end-user needs. Much research has been done into providing programmers with tools which separate the behavioural features of the software from the functional features [17]. Some examples illustrate the approach with respect to:

- *Distribution* — since the failure of distributed systems has very different failure semantics from centralised systems, the separation of this concern is not a simple task. Guerraoui [17] describes Garf, a software development tool which provides a library of abstractions to simplify distributed programming by enabling the programmer to develop the functionality of the software first, and then add distribution by using Garf.
- *User Interface Code* — the Chiron-1 user interface development system [36] introduces a series of layers that separate user interface code from application code by using user interface agents called artists which are attached to the abstract data types. Operations on the data types then trigger user interface activities.
- *User Manuals* — Thimbleby [37] developed Hyperdoc, a system which allows a programmer to develop the user manual alongside the user interface, so that the user manual mirrors the structure of the user interface.
- *Exception Handling* — Dellarocas [15] makes a case for separating exception handling from normal system operation. An exception handling service is provided for use by component developers, which uses a knowledge base to describe the failure of the system to the user.

- Kiczales [21] introduces *aspect-oriented programming.* Aspects refer to location, communication or synchronisation and once specified, are automatically combined with the application program by using some tool, like AspectJ [3]. Kersten and Murphy [20] have built a web-based learning environment using this programming paradigm.

All these examples require the programmer to specify the basic functionality of the application, and special concerns, albeit separately. Kiczales argues that this helps to reduce complexity that the programmer has to cope with.

Use of Tracking Information. Whereas the results of user interface monitoring are sometimes utilised by the end-user of a system [12], it is often done primarily for the benefit of system developers and maintenance teams. System resource monitoring is done exclusively for the benefit of system development teams. One important stake-holder in application usage, the end-user, is seldom catered for. Since the focus of this research is the end-user, and their feedback needs, I propose to provide the programmer with a feedback enhancing tool which will enable them to concentrate on the functionality of the program without the complication of having to provide extensive feedback. This will be done by means of tracking the application in order to provide the feedback. The author is unaware of any work which currently addresses augmenting feedback needs in this particular way.

1.2 Proposed Approach: Disentangling Feedback by Using Tracking

Tracking application activity is simple if the need is anticipated before, or during, the development of the system. The programmer can explicitly code the necessary reporting activities along with the coding of the application logic, and this tracking will be detailed and provide excellent reporting facilities. However, it does place a significant load on the programmer, and it is difficult to anticipate all tracking needs that could be required. This also does not solve the problem of tracking an *existing* application. It would be very useful to have an add-on generic facility available to all applications. This type of application tracking must, perforce, treat the entire application as a *black box*, thus providing very little detail about the internal application operation. The application's interaction with its environment — the user and the rest of the distributed system — will have to be observed, and these interactions tracked in order to provide a log of application activity. This will be done by inserting proxies to report on this activity. While this obviously does not provide the intricate detail which could be provided by internal application reporting, it does have the advantage of being generic enough to be applied to any application, and the added attraction of not requiring any special effort from the application programmer.
The research discussed in this paper was done in the context of end-user applications within Component Based Systems (CBSs) (Fig. 1).

- CBSs were chosen because they have unique characteristics which make such application tracking not only possible, but extremely beneficial. These characteristics include the independence of constituent components; the fact

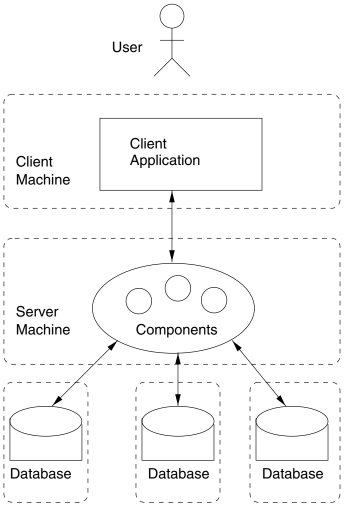


Fig. 1. CBS Application Architecture

that each component has its own documentation; and the use of interfaces to interact with these components. The need for such a facility is even greater in CBSs than in normal applications, due to this selfsame independent nature of the various component parts. This independence is a vital feature of CBSs since it allows the construction of a system from various parts to satisfy the user's exact requirements. An organisation can purchase different tailor-made components from various vendors, install them in a server runtime environment, and develop their end-user application to make use of these components.

However, the fact that the constituent parts are developed by largely unknown programmers means that the developer of the end-user application will have a limited understanding of the full functionality and expected behaviour of server components. An external tracking facility can therefore be extremely helpful to the programmer during both development and maintenance phases of an application, if informative feedback can be provided about application interaction with the CBS.

- The decision was made to concentrate on Java-based applications since Java has many features which suit tracking purposes admirably. Firstly, Java offers an introspective capability which allows us to dynamically explore details of Java classes. Secondly, the ability to dynamically substitute proxies is essential for external application tracking.

It should be stated at the outset that the aims in providing activity tracking are that it:

- should not interfere with the source code of the application;
- should not make any changes to existing package code, for any of the packages currently being used by the application;

– should require minimal application developer participation, and should even function adequately without it.

The mechanism for dynamically engaging the proxy between the user and the graphical user interface is explained in Sect. 2, while the mechanism for dynamically inserting the proxy between the client application and the rest of the CBS is explained in Sect. 3. The completed prototype is explained in Sect. 4. Section 5 discusses the advantages and disadvantages of the proposed approach. Section 6 discusses proposed future work on this project, and the paper is summarised in Sect. 7.

2 Interception of User Interface Communication

This section describes the mechanism used to insert a proxy, positioned as shown in Fig. 2, which tracks the appearance of, and interaction with, the user interface. The design of the user interface proxy is discussed in Sect. 2.2. How the proxy is used to satisfy our aims is discussed in Sect. 2.3. How information thus obtained is used to track user activity is discussed in Sect. 2.4.

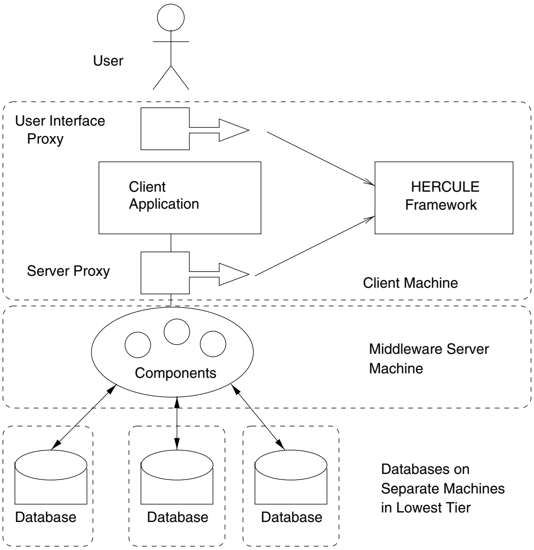


Fig. 2. CBS Application Architecture with Proxies

In order to track user activity, without, for the moment, being language specific, there are two requirements:

1. The first is to build an internal representation of the user interface structure. To do this, there is a need to know about each user interface component being created, and how the user interface is composed. For example, the window shown in Fig. 3 can be represented as a tree structure, as shown in Fig. 4.

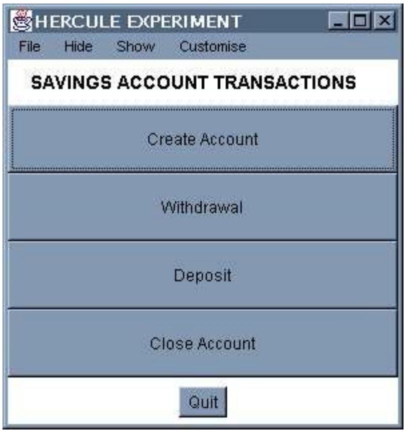


Fig. 3. The Client User Interface

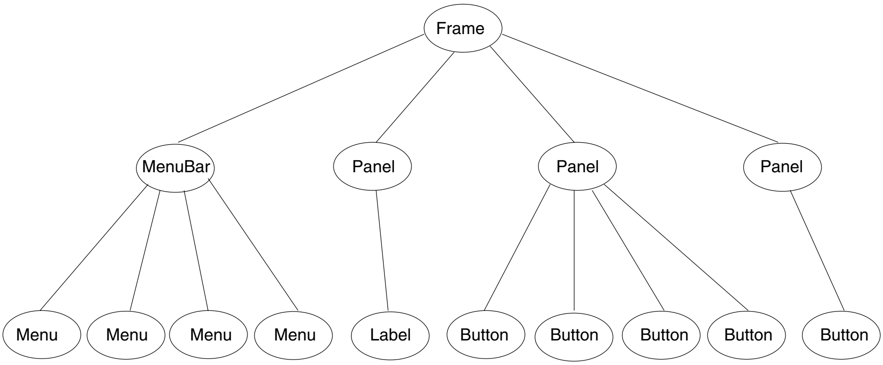


Fig. 4. The Internal User Interface Representation

2. The second is to keep track of user activities at the user interface, and to associate them with the parts of the interface being used. To watch user activities, a tracking facility needs to be notified whenever the user does something at the Graphical User Interface (GUI). Since GUIs are primarily event-based, we can reasonably expect to receive this information in the form of events. So, for example, in the `Window` shown in Fig. 3, the application obviously responds to button activations. In that case, HERCULE also needs to be apprised of button activations.

The user interface tree structure is required so that the user interface events make sense. Without such a structure, it would be impossible to identify windows containing components which have generated events, or to keep track of components within a window being added or removed, or to provide any sort of context sensitive feedback.

2.1 Inserting a Proxy — In Java

This section addresses Java specific issues with respect to inserting a proxy. Java is platform independent, so that a program written to run on one platform, using a particular Operating System (OS), can be executed on another platform using another OS, without alterations.

The way the Java Virtual Machine (JVM) provides this feature for the GUI is by means of a combination of the `java.awt.Toolkit` class and a library of platform dependent `Toolkit` classes. When the Java program instantiates user interface components in order to build a GUI, the actual component will use the `Toolkit` to establish a link to a platform dependent peer.

When the Java application interacts with these `java.awt` objects, the messages are relayed to platform dependent peers, in order to display the required GUI. The JVM handles these details so that the programmer is completely oblivious of the process. The programmer simply instantiates and invokes methods on the `java.awt` objects and subsequent calls to the peer objects are completely invisible. The `java.awt.Toolkit` class has the responsibility for loading these platform dependent classes. This `Toolkit` is loaded automatically by the `java.awt` classes when they are instantiated. A programmer will often never have to make direct use of this class at all. For example, they may do the following:

```
Button quit = new Button("Quit");
```

The `Button` class calls on the `Toolkit` to create a platform dependent peer object, `ButtonPeer`. This object is the actual platform specific object which is displayed on the user interface. If the programmer now calls:

```
quit.setLabel("Cancel");
```

then the `quit` object will call the `setLabel` method on `ButtonPeer` so that the label on the button on the GUI will change. The structure of this activity is shown in Fig. 5.

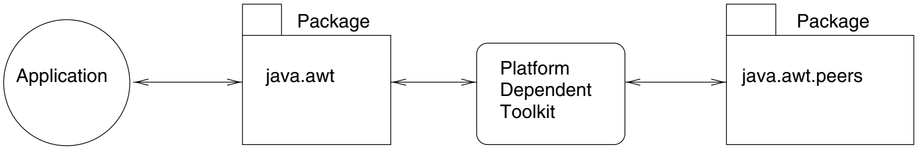


Fig. 5. The Use of the Toolkit to facilitate GUI platform independence

Our aim is to add activity tracking functionality to an application without making changes to either the application, or the `java` packages' source code. One approach would be to generate proxy classes for all the classes in the `java.awt` package, use an auxiliary class loader, and, by an additional level of indirection, substitute the proxy classes for the original classes. This satisfies our requirement that no part of the application should be altered, and it also does not interfere with the `java.awt` package. Unfortunately we cannot create a proxy for the `java.awt` package, because of the `java.awt.Toolkit` class. A proxy cannot be created for this class since it is abstract, and so the platform dependent `java.awt.Toolkit` and `java.awt` peers are loaded by the *application* classloader. This confuses the original classes which are loaded by their own separate classloader, and they consequently cannot reference the `Toolkit` [32]. Since the `java.awt` package is essential for our purpose in tracking user interface activity, another mechanism must be used.

The previous approach attempted to intercept user interface communications for *each* user interface component. However, an alternative source of information can be found in the `Toolkit` class, since Java requires the creation of all user interface objects be done via this class. Two factors make this a viable proposal:

1. The first is that `Toolkit` is an abstract class. In general, to obtain an instance of an abstract class, you either have to use an instance of a class that extends the abstract class, or use a static method which returns an instance of the class. The `java.awt` package makes use of the abstract class `java.awt.Toolkit`, which provides a static `getDefaultToolkit()` method, which gets the name of the platform dependent `Toolkit` class from system properties, and obtains an instance of that class from the OS libraries, to be returned to the caller.
2. The second, which relies on the first, is that the static `getDefaultToolkit()` method allows the use of an environment variable to specify which `Toolkit` is to be loaded. The `java.awt.Toolkit` incorporates a mechanism to allow the developer to substitute another `Toolkit` for the one which would, by default, be loaded by the JVM.

So, suppose a *proxy* `Toolkit` is written which extends `java.awt.Toolkit`, called `java.awt.ProxyToolkit`. The `EssentialApp` application can be told to use this proxy `Toolkit` by starting the application with the following command line:

```
java -Dawt.toolkit=java.awt.ProxyToolkit EssentialApp
```

The `java.awt.ProxyToolkit` is now instantiated when the application needs an instance of a `Toolkit`, and the proxy is thereby dynamically activated.

2.2 The ProxyToolkit

The `java.awt.ProxyToolkit` class extends `java.awt.Toolkit`. When the application calls the static `getDefaultToolkit` method to get an instance of the toolkit, the `ProxyToolkit` is created, and this `ProxyToolkit` then loads the OS specific `Toolkit`, so that the `ProxyToolkit` acts as a channel, relaying all calls to the platform dependent toolkit, and relaying all return values back to the application. The resulting structure is shown in Fig. 6.

Each method in the `ProxyToolkit` generates *construction* reports which relay information about the structure and composition of the user interface, enabling the construction of an internal structure duplicating each window structure. This structure provides the basis for making sense of user activity reports.

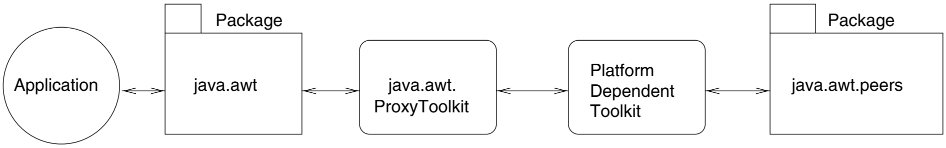


Fig. 6. The System using the ProxyToolkit

2.3 Watching User Activity

Once an internal structure has been created, the next requirement is to be able to keep track of user activities. This can only be done if we are informed when those actions occur. We could, upon learning that a component has been created, declare an interest in all events upon that component. This would mean that we would be interested in every button press, every mouse movement, every key press, window activation, and deactivation, and much more. This volume of reporting would slow the system unacceptably. The second best option is to register an interest in events which interest the application. These events would presumably precipitate some action on the part of the application, and are therefore meaningful activities from the point of view of the user when using that particular application.

All `java.awt` components allow other objects to declare an interest in events on the component by registering as a *listener*. The `ProxyToolkit` will be used to allow HERCULE to register an interest in events of interest, and thereby remain informed of all user activity at the user interface, by the receipt of *event* reports from the `ProxyToolkit`.

The event notifications received as a result of registering as a listener will serve to provide a tangible record of all user activity. Together with the previously defined internal structures representing these windows, the meaningful information can be provided about user interaction with the system.

2.4 Maintaining and Using the Internal Image of the GUI

The *construction* reports generated by the `ProxyToolkit` are used to build up a tree structure, depicting the appearance of the user interface, as shown in Fig. 4. *Event* reports will keep the tracking program informed of all activity, and it will know exactly what the user has been doing at any time, together with the effect on the user interface of that user activity [33]. A history of windows which are shown at the user interface is maintained, as well as a history of user actions which cause a change in the user interface appearance, including changes to "editable" entities like text fields.

3 Intercepting Server Communication

This section discusses the mechanism for inserting a proxy between the client application and the rest of the CBS, positioned as shown in Fig. 2. The figure depicts the CBS as a 3-tier system, which is common, but not essential for HERCULE's operation. A decision was made to insert proxies to intercept communication with the server components, because this would cause minimal interference with the application. This can also be done transparently, so that it requires no effort from the end-user or programmer.

3.1 Insertion of Proxies

Each server component interface is a potential source of contact with the component, and so each interface requires a proxy. The JVM makes use of a `CLASSPATH` environment variable that can be exploited to ensure that the JVM loads a *proxy* class *instead* of the original class, simply by putting the location of the proxy class ahead of the location of the original class in the `CLASSPATH`. For the purpose of the HERCULE prototype, we choose to implement a proxy insertion mechanism which would work for client applications using the JNDI package to contact the application server. JNDI requires the client application to establish communication with the server housing the server components before any connection can be made with those components. This connection is made by means of the `InitialContext` class. The proxy for this class will be dynamically inserted by making use of the above-mentioned `CLASSPATH` mechanism.

Once the proxy is inserted at this level, it is relatively simple to introduce proxies for each component, since the `InitialContext` object is used to locate required components, and the proxy object can be loaded and substituted for the original component completely transparently.

3.2 Using the Reports Generated by the Proxies

When HERCULE receives the reports from the proxies, they have to be stored so that the information can be retrieved at any time for feedback purposes. It is important to realise that the server proxies are totally unaware of the user interface proxy, and that they therefore have no communication with one another. The only way that HERCULE can link user actions to server method invocations is by using the time factor enclosed within the generated reports. Therefore, when server reports are received, these actions need to be linked to the user actions which preceded them. At this stage, there is a need to clarify the strategy for mapping these separate activities to each other.

Since not all user activity will result in server component method invocations, there could be a number of user actions occurring before a method invocation. In the same way, a whole string of method invocations could be precipitated by a sequence of user actions. In each application thread, the sequence of user actions which precede one or more method invocations is called a *UA-sequence* (User Action sequence), and the series of server calls thus precipitated is called an *MI-sequence* (Method Invocation Sequence). When a UA-sequence is matched to an MI-sequence, we can call this mapping an *Episode*. This is illustrated in Fig. 7.

When storing the proxy information (both user interface and server), it is vital to store it in the form of Episodes, which can then be depicted in the

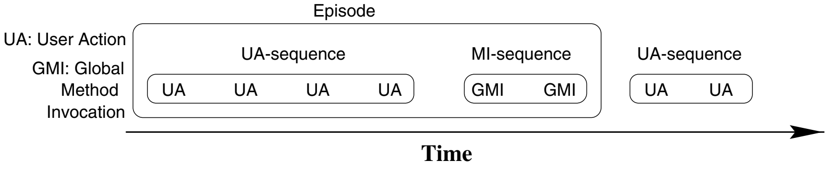


Fig. 7. UA-sequences, MI-sequences and an Episode

feedback. It is still necessary to keep them apart for some specialised feedback requirements, so HERCULE will store a list of UA-sequences, and link each UA-sequence to the MI-sequence precipitated by the UA-sequence. These two lists will be linked one to the other, forming a history of session Episodes.

4 HERCULE Implementation

In order to test the viability of this approach, a prototype of HERCULE was implemented. The prototype was tested on a three tier CBS with *Enterprise Java Beans* (EJBs) [27] fulfilling the role of the server components. The application server used was the Tengah server from Weblogic [38], an all-Java application server. The test system was composed of a client on an NT host running on an Intel, the Tengah server running on Solaris on an Intel, with the third level being made up by a Cloudscape database [40] containing a set of client accounts.

HERCULE tracks application activity by dynamically inserting proxies, and extracts information based on the reports generated by these proxies. The next section discusses the inputs that HERCULE receives from the two proxies. Section 4.2 explains how HERCULE is customised for any given EJB, and how HERCULE extracts the required inputs at runtime. Section 4.3 shows how HERCULE would be used with a running application, and Sect. 4.4 gives an example of how the information gleaned from the proxies can be presented to the user in order to provide a useful tool to programmers, end-users and support staff. Section 4.5 reports on a preliminary study of how the presence of HERCULE affects application performance.

4.1 Inputs into HERCULE

HERCULE basically operates based on two types of inputs. The first is made up of the documentation and Java class files delivered with the EJB. The second comprises the reports generated, at runtime, by the proxies. The following documents are the least we would expect to be delivered along with each server component, since they provide the basic minimum information required to use the component:

1. An *Application Programmer Interface* (API) document, explaining the purpose of the component, and giving details of method functionality, for example, `javadoc` [26] output.

2. One or more interfaces through which the component can be accessed.
3. A deployment document which specifies the context dependencies of the server component, and explains how the component should be deployed.

HERCULE must use the information from this documentation to customise itself. This customisation facilitates the operation of the proxies at runtime. HERCULE receives two types of reports from run-time invoked proxies:

1. *User Interface Reports*: signaling events and the user interface construction. These events enable HERCULE to keep a history of user interface appearance and user activity.
2. *Server Method Invocation Reports*: The reports received here indicate different stages of communication, including initial establishment of communication with the server and global method invocations following thereafter.

Once the UA-sequences have been linked to the MI-sequences, and the Episodes constructed, the results need to be depicted in a helpful manner on the screen. There are many aspects of this interaction that could be depicted, but for HERCULE, the decision was made to depict the success or failure of each Episode. This decision was made because our focus is to provide end-user feedback, and the success or failure of an Episode is of critical interest to the end-user. Since a particular application session could easily generate many Episodes, the display chosen has some important characteristics:

- it should be able to depict either one or many Episodes in a clear manner, so that the user can obtain as much information as possible at a glance;
- it should not intrude, but offer user assistance. Thus, it should use as little screen space as possible. Many feedback devices tend to become overpowering, and the last thing we want to do is to annoy; and
- it should allow the user to step backwards in time to view and confirm their previous actions.

4.2 HERCULE's Phases

HERCULE has two distinct phases of use, discovery and runtime. The discovery phase is a *customisation* phase, which serves to inform HERCULE, essentially a generic feedback mechanism, of the server components which will be used by an application. During the *runtime* phase, the results of the customisation will be used to facilitate the required feedback.

The customisation is done prior to HERCULE being used, and as often as necessary after that as the programmer becomes more familiar with the operation of the component. HERCULE makes use of the server component documentation to customise the framework for a particular server component. The component documentation is "mined" in order to extract *descriptor objects* which hold semantic details about the methods used to access the server components, and to generate proxies. If we consider these documents to be the *input* to the discovery process, then the output we produce is:

- *proxy classes* used in intercepting communication with the server.
- *descriptor objects*: which are essential to the visualisation of session activity. Tracking will only be meaningful if its results can be depicted in an information-rich and useful fashion. In order to provide the users with explanations of server activity, the method invocations should be described in terms easily understood by the user, rather than in language familiar to the programmer of the system. These explanations are all to be found in the server component API documentation, and the initial descriptor objects will thus be derived from these documents. Since Java class documentation is generally produced by `javadoc`, this makes the mining process simpler[2]. This mining process should produce at least an *adequate* descriptor object, since it contains the information as obtained from the API document. In order to improve this object, HERCULE provides a tool to allow the programmer to augment the descriptor object. With the programmer's assistance the descriptor object can be augmented from *merely adequate*, to *substantially helpful*.

4.3 HERCULE in Action

An example of how HERCULE would be used by the user of an applicationn will now be given. HERCULE runs in a separate process so that its execution and termination are not dependent on the application. The HERCULE console operates in the following modes:

- *waiting mode* as it waits for the application proxies to make contact.
- *dynamic feedback mode* after the application has made contact via the proxies, and while it executes.
- *static feedback mode* after HERCULE registers the termination of the application but stays active so that the user can use the console to provide post-execution feedback, if required.

We will consider the case of two different users — the programmer and the end-user.

The Programmer. The programmer will run the discovery process to customise HERCULE for the server components which will be used by the application. Descriptor objects and proxies will be generated for each interface of each component. These will automatically be compiled and made available to the JVM.

The programmer would then start execution of HERCULE, and the HERCULE console would appear on the rightmost side of the screen. An icon would appear at the base of the screen if it is being run on a Windows platform.

The programmer now starts execution of their program. HERCULE tracks the application activity and uses the generated descriptor objects to provide an

[2] If this is not done by `javadoc`, it becomes difficult to mine since we have no idea how the documentation would be structured. The next EJB specification requires the use of XML for this documentation, which would make the process even simpler.

explanation of method invocations. If the programmer needs to augment these, the discovery process can be executed again, and the explanations changed. If a method invocation could result in an exception being thrown, the programmer can also enter meaningful explanations for these exceptions.

Once the application program shuts down, the traffic lights display on the HERCULE console will display a message indicating that the application has finished executing. This is done so that the information about the session is still available even if the application crashes. The programmer will then have a tangible record of inputs given, together with details of method invocations and results returned by the server components.

Should the programmer want to track a new session, HERCULE can be reset by choosing a `reset` option from the console menus.

The End-User. The end-user starts execution of HERCULE, and the HERCULE console appears on the rightmost side of the screen.

HERCULE tracks the application and displays the Episodes dynamically. If the user chooses not to actively use HERCULE, it will remain unobtrusively in the background. If the user needs an explanation, HERCULE can be made active, and will provide the user with an explanation. Should this still be inadequate a support person can be summoned, and the record of activities and explanations can be used to solve the problem.

4.4 Feedback Provided

In Sect. 4.1 some criteria for the feedback display were mentioned. The console designed for HERCULE was created with those requirements in mind, and satisfies them as follows:

- it depicts the all Episodes for the entire application session in one window;
- it allows detailed information about Episodes to be obtained at the click of a mouse;
- it depicts a great deal of information in a small screen space;
- it does not intrude, but is always available as an icon, offering the possibility of obtaining feedback at any time; and
- it allows the user to obtain information about previous Episodes at the click of a mouse.

At runtime, the HERCULE console (Fig. 8) provides the following information, which is dynamically updated as the user works:

1. A traffic lights widget depicting the current system state. This will display:
   - red when the application cannot be tracked. The legend beside the traffic light will display the result of HERCULE's attempt to diagnose the cause;
   - orange when a component is busy servicing a request; and
   - green when HERCULE is in dynamic feedback mode.

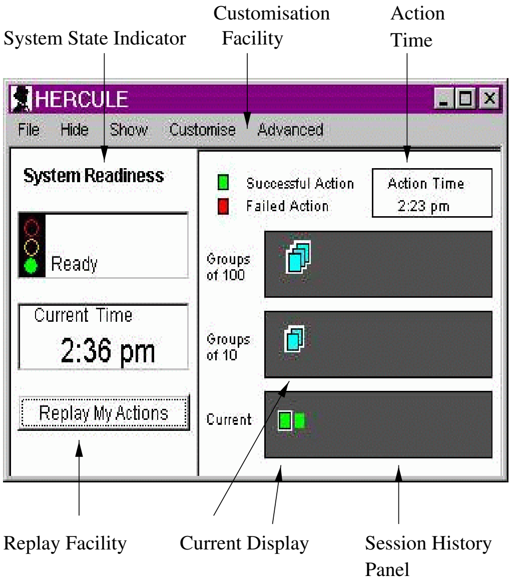
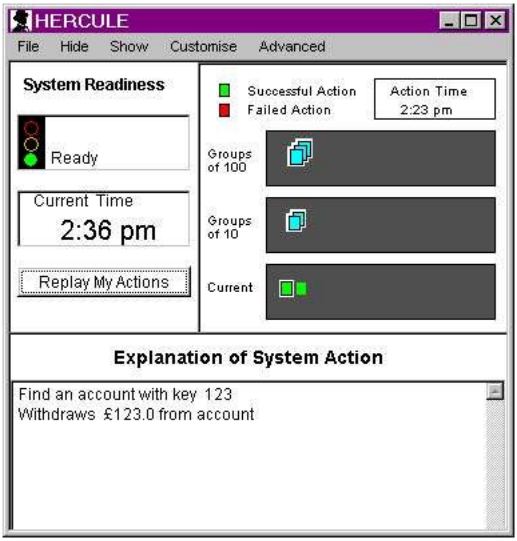


Fig. 8. The HERCULE Console

2. A `Replay My Actions` button will summon a playback facility which allows the user to view a screen replay of all UA-sequences as they took place. This is done in the form of the windows displayed by the application being shown to the user, one at a time, with a highlight on the action which caused the transition to the next window. For example, in the Window in Fig. 3, if the user clicked on the `Create Account` button, that button would be highlighted by setting the background colour to yellow in the replay window. This serves to remind the user of past actions. By providing this functionality, HERCULE supports users by alleviating their weaknesses (such as limited working memory), and capitalising on, and utilising their strengths (such as swift pattern recognition, and retrieving relevant information about the meaning of these patterns quickly). This replay has no effect on the application whatsoever, in accordance with the non-intrusion policy, and should be considered to be rather like an action replay used in television sports broadcasts.
3. A session history panel which presents all Episodes, displayed in three separate panels:
   - the bottom panel displaying the last ten Episodes,
   - the middle panel depicting groups of ten Episodes, and
   - the top panel depicting groups of hundreds of Episodes.

   Each distinct Episode is displayed as a coloured rectangle. This depicts the result of the *MI-sequence* resulting from the Episode UA-sequence as:
   - red if it failed — assumed if the server throws an exception,
   - yellow if the outcome is pending, and
   - green if it succeeded — assumed by the absence of an exception.



Find an account with key 123
Withdraws £123.0 from account

Fig. 9. The User Viewing an Explanation of a Previous Episode

4. In order to provide for different types of user needs, HERCULE can be customised to give extended feedback. The type of user feedback needs already identified are:
   a) explanation of an MI-sequence. In other words, what the system did as a result of a UA-sequence. This is primarily an end-user need, and is met as shown in Fig. 9.
   b) assistance in debugging. A detailed explanation of the method invocations making up the MI-sequence, together with the parameters for each invocation, and the return value or exception thrown.
   c) performance monitoring, as shown in Fig. 10. For example, a graphical display of response times is a good indicator of application performance with respect to communication with the rest of the CBS. This feedback component could be of use to system support.

   There are two aspects of these feedback components to be considered:
   a) *Which Episode the feedback applies to* — The first and second of the above-mentioned feedback needs should be met by displaying an explanation of the most recent Episode, or a previous one, as required by the user. Feedback should be provided for the most recent Episode by default, allowing the user to request the display of the explanation of a previous Episode by clicking on one of the previous blocks in the lower panel. Feedback can be obtained for Episodes not shown in the lower panel by clicking on one of the grouped symbols, as depicted by groups of blocks in the middle and upper panels. You will note from Fig. 8, that each panel displays some rectangles, and that one is highlighted. The highlighted rectangle indicates the Episode for which feedback is currently being given in the visible feedback components. The *Current*

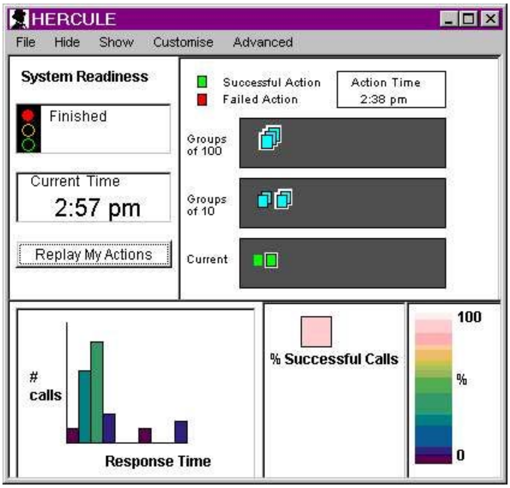


Fig. 10. The HERCULE Console showing the Support Panel

Display label in Fig. 8 points to the highlighted rectangles in the history panels, indicating that the second last Episode MI-sequence explanations would be displayed (if any feedback components were visible).
The user's immediate feedback requirements with respect to individual Episodes, as indicated by clicking on a block which represents a previous Episode, will be met by dynamically reflecting the feedback for that Episode in the information displayed by each of the visible feedback components. On the console shown in Fig. 9, the Episode actions are explained as `Withdraws 123.0 from account`. This is not the explanation of the most recent Episode MI-sequence, since the rectangle in the last but one position is highlighted, indicating that the explanation belongs to the second last Episode.

b) *How additional feedback components are plugged into the system* — The HERCULE console is dynamically extensible, so that the identification of a new user feedback need can be accommodated. New HERCULE feedback components can be coded, and plugged into the HERCULE console at runtime. The top section of the console, as shown in Fig. 8, will always be displayed, since it provides the core functionality of the console display. When a programmer wants to add a new feedback component, the `Advanced` menu is chosen, and the programmer clicks on `Plug new Component in`. The class name of the new component is entered, and HERCULE will register the existence of this component. The user can choose which components should be visible or invisible by using the `Show` and `Hide` menus, and the user can save their overall preferences with respect to visibility of tailored feedback components, so that the console is customised to satisfy their particular needs. The `Customise` menu is reserved for customisation such as user preferences with respect to feedback components displayed, and choice of whether a failed episode should be brought to the user's attention by an audio signal or not. To support this extensibility of the HERCULE console, the following are provided:

- An abstract class named `HerculeComponent` (which extends `java.awt.Panel`). This class must be extended by any feedback component to be incorporated into the HERCULE console.
- A `HistoryListener` interface. The feedback component implements this interface, and registers as a listener with the history panel. The feedback component will then be notified of user actions at the history panel, which will enable it to provide relevant feedback. The feedback component will implement this interface if it is going to provide dynamic feedback related to a specific Episode.
- An `OutcomeListener` interface. The feedback component implements this interface, and registers as a listener with the history panel. The feedback component will then be notified of the outcome of Episode MI-sequences. The feedback component will typically implement this interface if it wants to provide statistics about the number of successful Episodes, or performance.

4.5 Performance Reduction

It is very important that the presence of HERCULE should not affect application performance unacceptably. Since HERCULE inserts proxies between the user and the user interface, and between the user and the rest of the CBS, we can expect any performance degradation to take place:

1. when the `Toolkit` is being loaded, since an extra level of indirection is being introduced;
2. when the initial connection with the server is being forged, since this is where the server proxy will be introduced;
3. whenever a new window is being constructed; and
4. when global methods are invoked on distributed components. Two types of methods need to be considered independently, methods which will require action by the component container, and methods on the component itself. The former take longer than the latter to process.

A preliminary study of performance differences was undertaken, by running the example application twenty times both with and without HERCULE. Where effects were observed, the results are shown in the table below[3].

Action	Without Proxies	With Proxies
Display of Initial Application Window	1.44	2.45
Initial Contact with Server	5.92	8.73
Container Method Invocation	1.40	1.56
Component Method Invocation	0.25	0.54

[3] There was no discernable effect when new windows were constructed, with only the time taken for the initial window being affected.

It is clear that the user *will* have to pay a penalty for using HERCULE. It would be unreasonable to expect otherwise. The entire insurance industry is based on the "present pain, future gain" principle. Shneiderman [34] cites research which shows that modest variability in response times is deemed by users to be acceptable. If the user sees the benefits of using HERCULE, they will hopefully be prepared to pay the small penalty of slightly longer response times, for the future gain of having informative and extensive feedback available. Finally, it can be expected that the negative impact of HERCULE will soon become less apparent due to the increased efficiency and speed of hardware.

5 Pros and Cons of this Approach

End-user feedback needs are often not adequately catered for. Traditional approaches to providing feedback have required the application programmer to incorporate it in the end-user application, or to provide it by means of an add-on facility like an online manual. Manuals cannot hope to supply dynamic feedback, but at best can only provide static help and explanations. The only other way to provide feedback is by means of a set of libraries somewhat like those provided by the Garf tool [17], which a programmer could use to provide feedback. Feedback is quite unlike distribution, however, since it is far more demanding and pervasive than distribution. The guidelines for providing feedback advise that *every* operator action should have accompanying feedback [34], and that the user should be continuously informed [2, 28], which means that the load on the programmer is substantial. The advantages of the HERCULE approach are that:

– it requires minimal participation from the programmer;
– it frees the programmer from a heavy load of catering for feedback continuously;
– it can be disabled or enabled as required, since it does not require the addition of any code to the application;
– it provides a uniform, customisable feedback for different applications;
– the extensibility of the HERCULE console makes it easy to accommodate changing user needs; and
– the tendency of distributed systems to indeterminate failures makes it useful to have a standard way of indicating that an error has occurred, and for explaining the causes.

The disadvantages of the approach are that:

– it can only give feedback based on the external activities of the application. Thus the feedback that can be provided is limited to the interaction of the application with the user and the rest of the distributed system; and
– it requires the use of a language with introspective capabilities, since this is essential for the generation of proxies.

The HERCULE approach was chosen since the primary aim was to simplify the programmer's task of feedback provision. This had to be done with minimum disruption to the normal application development. The other important aim was that this facility should be easily disabled or enabled. HERCULE satisfies these requirements.

6 Future Work

An Episode has been defined as a UA-sequence followed by an MI-sequence, and HERCULE presently links the UA-sequences to the precipitated MI-sequences by using the time in each report. This will obviously cause problems in multi-threaded applications. Work is underway to design and implement a version which will cater for multi-threaded applications. We are currently further evaluating the negative impact of HERCULE on application performance. We are also planning user evaluation trials of HERCULE to measure user reactions to it. The querying capabilities of the console are presently limited to stepping through the Episodes one at a time. It would be beneficial to extend the system to group like Episodes to enhance querying facilities.

7 Conclusions

This paper has shown that it is indeed possible to augment the feedback provided by an application by tracking application activity and providing the feedback by means of a generic framework. The HERCULE framework uses the information gained from this tracking to provide different users of the application with a visualisation of their session activity. The scheme has been proved to be viable by means of the implementation of the HERCULE prototype.

The mechanism for dynamic insertion of proxies detailed in this paper are necessarily applicable only to Java systems using JNDI. That does not mean that it cannot be done in other application architectures. Insertion of the user interface proxy is done relatively simply in Windows systems, by means of hooks explicitly provided by the Microsoft OS. The mechanism for insertion of the server proxy is easily extended for other types of CBS architectures, since communication protocols in these systems are fairly standardised. For example, there are a few techniques which will cover insertion of server proxies for most CBSs:

- COM+, the Microsoft component model, and CORBA (the OMG model) allow the specification of interceptor components between a client application and server components [14].
- Browsers (often housing client applications) commonly make use of proxies, and many generic proxies, which can be tailored to specific needs, are widely available.
- Two widely used protocols would allow the insertion of a proxy using the `CLASSPATH` mechanism as follows:
  - The Java Naming and Directory Interface(JNDI) protocol — already explained in Section 3.1.

- The Remote Method Invocation (RMI) protocol — The `Naming` class is used to establish contact with services offered on any machine, and so a proxy `Naming` object can be used as the point of insertion.

There are some issues which need to be considered when advocating a scheme such as the one outlined in this paper. The first is whether the support structure for applications should allow the normal execution environment to be replaced by one which reports on user activities. The second issue raises the question of whether the end-user will resent HERCULE — seeing it as some sort of spying device — or whether it will be perceived as a worthwhile feedback provider. These issues are outside the scope of this paper, but nevertheless constitute an interesting research area.

Acknowledgements. Special thanks to my supervisor, Richard Cooper, for his help and guidance. Thanks too to Huw Evans and Ela Hunt for their extremely helpful comments on the draft of this paper. I also acknowledge the valuable contributions of James Begole and Susan Spence. This research is supported by a scholarship from the Association of Commonwealth Universities and a grant from the Foundation for Research and Development in South Africa, and the University of South Africa. I am currently on leave of absence from the University of South Africa and I would like to acknowledge their magnanimity in allowing me this extended period of absence.

I gratefully acknowledge the donation of the Tengah server from Weblogic/BEA (`URL: weblogic.beasys.com`), and express my appreciation for their prompt responses to my queries. This work could not have been carried out without their kind donation.

References

[1] ACM. *1998 International Workshop on Component-Based Software Engineering. Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Kyoto, Japan, April 25–26, 1998 1988. URL: http://www.sei.cmu.edu/cbs/icse98/papers/index.html

[2] *Human Interface Guildelines: The Apple Desktop Interface.* Addison-Wesley, Reading, Massachusetts, 1987. Apple Computer Inc.

[3] http://www.parc.xerox.com/spl/projects/aop/aspectj. AspectJ Web Page, 1998.

[4] T. Ball and J.R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, 1994.

[5] L. Blackshaw and B. Fishhoff. Decision making in online searching. *Journal of the American Society for Information Science*, 39:369–389, 1988.

[6] N. Borenstein. *Programming as if People Mattered.* Princeton Univeristy Press, Princeton, New Jersey, 1991.

[7] A.N. Burton and P.H.J. Kelly. Workload characterization and using lightweight system call tracing and re-execution. In *IEEE International Performance Computing and Communications Conference. IPCCC '98*, Phoenix/Tempe, Arizona, USA, February 16–18 1998. IEEE.

[8] A.N. Burton and P.H.J. Kelly. Tracing and reexecuting operating system calls for reproducible performance experiments. *Computers and Electrical Engineering: An International Journal*, May 1999.

[9] F.R. Campagnoni and K. Ehrlich. Retrieval using a hypertext-based help system. *ACM Transactions on Information Systems*, 7:271–291, 1989.

[10] J.M. Carroll, editor. *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction.* MIT Press, Cambridge, MA, 1987.

[11] J.M. Carroll and M.B. Rosson. The paradox of the active user. In *[10]*, chapter 5, pages 80–111. 1987.

[12] M. Chalmers, K. Rodden, and D. Brodbeck. The order of things: Activity-centred information access. In *Proceedings of the 7th International Conference on the World Wide Web*, pages 359–367, Brisbane, Australia, Oct 5-7 1998.

[13] H.C. Chan, K.K. Wei, and K.L. Siau. The effect of a database feedback system on user performance. *Behaviour and Information Technology*, 14(3):152–62, 1995.

[14] D. Chappell. Com+. WEB Document, April 1998. www.chappellassoc.com/articles.htm.

[15] C. Dellarocas. Toward exception handling infrastructures for component-based systems. In *[1]*, 1998.

[16] T. Fawcett and F. Provost. Activity monitoring: Noticing interesting changes in behaviour. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, CA USA, August 15–18 1990. ACM.

[17] R. Guerraoui, B. Garbinato, and K.R. Mazouni. Garf: A tool for programming reliable distributed applications. *IEEE Concurrency*, 5(4):32–39, October/December 1997.

[18] C. Jeffery, W, Zhou, K. Templer, and M. Brazell. A lightweight architecture for pro- gram execution monitoring. In *ACM SIGPLAN/SIGSOFT workshop on Progra Analysis for Software Tools and Engineering*, Montreal, Canada, June 16 1998. ACM.

[19] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.

[20] M.A. Kersten and G.C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. Technical Report TR-99-04, Department of Computer Science, University of British Columbia, March 31 1999. Wed, 07 Apr 1999 21:31:26 GMT.

[21] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4es):154–154, December 1996.

[22] P. Leibscher and G. Marchionini. Browse and analytical search strategies in a full-text cd-rom encyclopedia. *School Library Media Quarterly*, Summer:223–233, 1988.

[23] C. Lewis. Understanding what's happening in system interactions. In D.A. Norman and S.W. Draper, editors, *[30]*, chapter 8, pages 171–186. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[24] X. Lin, P. Liebscher, and G. Marchionini. Graphical Representations of Electronic search Patterns. *Journal of the American Society for Information Science*, 42(7):469–478, 1991.

[25] G. Marchionini. Information-seeking strategies of novices using a full-text electronic encyclopedia. *Journal of the American Society for Information Science*, 50:54–66, 1989.

[26] Sun Microsystems. javadoc - The Java API documentation Generator. Web Document. http://java.sun.com/products/jdk/1.3/docs/tooldocs/solaris/javadoc.html.

[27] Sun Microsystems. Enterprise Java Beans Specification. Web Document. URL: java.sun.com/products/ejb, March 1998.

[28] J. Nielsen. *Usability Engineering*. AP Professional, Boston, 1993.

[29] D. Norman. The “problem” of automation: Inappropriate feedback and interaction, not “overautomation”. Technical Report ICS Report 8904, Institute for Cognitive Science, University of California, San Diego, La Jolla, California, 92093, 1989.

[30] D.A. Norman and S.W. Draper, editors. User Centred System Design. *New Perspectives on Human-Computer Interaction*. Lawrence Erlbaum Associates, Publishers, Hilldale, New Jersey, 1986.

[31] J.R Olsen. Cognitive analysis of people’s use of software. In *[10]*, chapter 10, pages 260–293. 1987.

[32] K.V. Renaud. A Non-Invasive Mechanism for Monitoring Calls to Java Packages. Technical Report TR-1999-32, Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, G12 8RZ, April 1999.

[33] K.V. Renaud. Tracking activity at the user interface in a Java application. Technical Report TR-1999-33, Department of Computing Science, University of Glasgow, 17 Lilybank Gardens, Glasgow, G12 8RZ, April 1999.

[34] B. Shneiderman. *Designing the User Interface*. Addison-Wesley, Reading, Massachusetts, 1998.

[35] M. Siegle and R. Hofmann. Monitoring program behaviour on SUPRENUM. In *International Conference on Computer Architecture. Proceedings of the 19th Annual International Symposium on Computer Architecture*, Queensland, Australia, May 19–21, 1992 1992. ACM.

[36] R.N. Taylor and G.F. Johnson. Separations of concerns in the Chiron-1 user interface development and management system. In Stacey Ashlund, Ken Mullet, Austin Henderson, Erik Hollnagel, and Ted White, editors, *Proceedings of the Conference on Human Factors in computing systems*, pages 367–374, New York, 24–29 April 1993. ACM Press.

[37] H. Thimblebey. Combining systems and manuals. In J.L. Alty, D. Diaper, and S. Draper, editors, *People and Computers VIII HCI’93*, pages 479–88, 1993.

[38] A. Thomas. Selecting Enterprise JavaBeans Technology. Prepared for WebLogic, Inc., July 1998. http://www.beasys.com/products/weblogic/server/papers.html.

[39] J.G. Trafton and D.P. Brock. Simplifying interactions with task model tracing. ACT-R Summer School, Psychology Department, Carnegie Mellon University, June 1996.

[40] S. Willett. Cloudscape Woos VARs for Java Database. *Computer Reseller News 6-99*, June 1999. http://www.crn.com/search/display.asp?ArticleID=7017.

[41] D. Wybranietz and D. Haban. Monitoring and performance measuring distributed systems during operation. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 197–206, Santa Fe, USA, May 1988. ACM.

Automated Test Case Generation from Dynamic Models

Peter Fröhlich[1] and Johannes Link[2]

[1] ABB Corporate Research Center, Speyerer Straße 4,
D-69115 Heidelberg, Germany
peter.froehlich@de.abb.com
[2] Andrena Objects GmbH, Albert-Nestler-Straße 9, D-76131 Karlsruhe, Germany
johannes.link@andrena.de

Abstract. We have recently shown how use cases can be systematically transformed into UML state charts considering all relevant information from a use case specification, including pre- and postconditions. The resulting state charts can have transitions with conditions and actions, as well as nested states (sub and stub states). The current paper outlines how test suites with a given coverage level can be automatically generated from these state charts. We do so by mapping state chart elements to the *STRIPS* planning language. The application of the state of the art planning tool *graphplan* yields the different test cases as solutions to a planning problem. The test cases (sequences of messages plus test data) can be used for automated or manual software testing on system level.

1 Introduction

The systematic production of high-quality software, which meets its specification, is still a major problem. Although formal specification methods have been around for a long time, only a few safety-critical domains justify the enormous effort of their application. The state of the practice, which relies on testing to force the quality into the product at the end of the development process, is also unsatisfactory. The need for effective test automation adds to this problem, because the creation and maintenance of the testware is a source of inconsistency itself and is becoming a task of comparable complexity as the construction of the code.

To trace requirements throughout the software's life cycle - testing included - seems to be an appropriate way around the above mentioned problems. It can minimise the deviation of the work products from the specification. To maintain consistency throughout the activities of the software process, it has to be defined how each part of the requirements specification impacts each of the downstream artifacts. This strong traceability can already be achieved for certain parts of the requirements. For example, steps in a use case relate to messages in dynamic UML models [19], which map (in some contexts) to methods of a programming language and to steps in a test case. For other parts, the systematic exploitation of the information throughout all phases in the software process is not yet defined.

In the current paper, we describe a full mapping of all elements of a typical use case document [3] to a UML state machine. Based on this state machine we afterwards apply AI planning methods to derive test suites with a given coverage level. The test generation method takes into account the specific elements (conditional

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp.472–491, 2000.
© Springer-Verlag Berlin Heidelberg 2000

transitions, nested states) resulting from our use case to state machine transformation. We do not know of any other approach to systematically transform use cases via dynamic UML models (part of the software analysis and design and test model) into test suites, which considers the full amount of information provided by a use case document. We thus provide a method which takes all the information from the requirements, incorporates the necessary generalisations made during the analysis phase (represented in a state chart), and enables the verification of the requirements at the end of the software process.

The paper is structured as follows: Section 2 discusses related work on test generation from state machines. Section 3 is concerned with the first part of the transformation, i.e. the generation of a state machine based on a use case document; this section is an abridged version of our discussion in [7]. In section 4 we then focus on how the state machine created from the use cases can be used for test generation. We discuss the relationship between the test generation problem and *STRIPS* planning [5]. We define an algorithm for test suite generation and study an example using the graphplan [2] tool. The appendix provides the *graphplan* input and output for the example discussed in the text.

2 Related Work

There is common agreement in literature, e.g. [11], that use cases should be used for deriving test cases. However, concrete approaches how to do that are rare. Jacobson suggests in [11] three general kinds of tests that can be derived from use cases: (1) tests of the expected flow of events, (2) tests of unusual flows of events and (3) tests of any requirement attached to a use case. Jacobson does not go into the details of how to choose test cases and how to know when you are done.

Binder proposes in [1] to enhance use cases with a few “testability extensions”: the domain of each participating variable, the required input/output relationships among use case variables, the relative frequency of each use case and the sequential dependencies among use cases. He uses these extensions to describe in detail how to develop test cases for use cases and the order in which they should be run.

Unlike use cases state diagrams have long been considered as important for the development of software tests [1][7][13]. One recent discussion can be found in [13]. Therein Marick discusses both simple state machines and state charts as sources for tests. The construction of state charts is described more informally, compared to our method in section 3. The author starts from single scenarios instead of use cases; in this way he has no systematic information on the relationship among the scenarios. Pre- and postconditions of the scenarios as provided by our use case template are not considered. While we use nested states to systematically capture the different levels of sceanarios, Marick [13] uses them informally to group similar states in the state machine. The method for deriving test cases described by Marick takes into account important features of state charts but is described informally, compared to our semi-automatical approach using AI planning methods. The coverage level targeted by Marick's method is to exercise every transition. Our formal approach introduced in section 4 achieves the same coverage level as Marick's manual method.

3 From Use Cases to Dynamic UML Models

Use Cases [11] are a popular formalism for capturing functional requirements and business requirements. Use cases are a good means to communicate with a customer. They are the unit of work in incremental object-oriented software processes like the Rational Unified Process [12]. Furthermore, they are a good basis for systematic testing. In [19] Rumbaugh et al. define a use case as "the specification of sequences of actions, including variant sequences and error sequences that a system, subsystem or class can perform by interacting with outside actors”. While the advantages of use cases for requirements engineering are widely accepted, the impact of use cases on software design is less clear. The UML meta model [15] offers three different notations for designing dynamic system/subsystem/class behaviour based on use cases, as discussed in [20]:

- *Activity diagrams* interpret use cases as branching processes. Some authors [21] recommend their use for the formalisation of use cases. However, doing this has considerable disadvantages: First, state diagrams correspond directly to the object-oriented paradigm, whereas activity diagrams model the control flow of a program. Thus they provide the same means for creating a spaghetti control-structure as do flow diagrams. Second, the distinction between normal and abnormal behaviour, which is one of the benefits of use case analysis is lost.
- *Interaction diagrams* (sequence diagrams and collaboration diagrams) can be used to formalise single scenarios contained in use cases. In [18] Rumbaugh proposes to start dynamic modelling with scenarios in text form, formalise them as sequences of events in interaction diagrams and merge them subsequently into state diagrams showing the complete lifecycle of an object. This seems to be a rather unnatural approach for formalising use cases. Since a use case is a hierarchical collection of scenarios, these have to be formalised separately in interaction diagrams and then merged again into a single state diagram. [8][18][19][4]
- *State machines* specify the behaviour of a system/subsystem/class in reaction to events from actors. In contrast to interaction diagrams they visualise multiple scenarios, e.g. the hierarchy of scenarios described by a use case (see section 3.1).

3.1 Use Cases to State Machines

This section briefly describes a transformation from use cases to state machines, which we describe more thoroughly in [7]. State machines and state diagrams have a long history in computer science. Recent versions of UML [15] include an expressive state diagrams concept inspired by Harel's state charts [8][9][10]. Especially the abstraction mechanisms in the UML state machine formalism, i.e. nesting of states and stubs, allow us to map all the important elements of our use case documents to state machines.

Use Case Documents

The following simple example describes, how a library user borrows a book. The structure of the use case was inspired by Alistair Cockburn’s use case template [3]. This template describes the scenarios of the use case in a hierarchical fashion. The

Main Success Scenario is the straightforward sequence of steps leading to the achievement of the user's goal without consideration of possible problems. With each step, possible error situation and their resolution can be described in the *Extensions* section. Further, there may be different alternative ways to execute a step (e.g. search a book by title or search by author). These alternatives are described in the *Variations* section. The *Preconditions* section captures constraints, which the state of the world must satisfy before the use case can be executed. These are typically properties of the user (e.g. the user must have an account) or the state of program execution (e.g. the user is logged in). The *Postconditions* section on the other hand describes the conditions, which the use case establishes. Thus, Pre- and Postconditions together define the *contract* of the use case [14]. A step in the use case can refer to another use case being called. In the *Borrow Book* example below, step 8a1) references the *Log in* use case. The referenced use cases are summarised in the *Included Use Cases* section.

Name	Borrow Book
Goal	This use case describes how a library user selects and then borrows a book from the library.
Preconditions	None
Postconditions	The user is registered as the borrower of the book in the library system.
Main Success Scenario	1. The user selects the search function from the main menu. 2. The system displays the search form. 3. The user enters the title of a book (possibly using wild-cards). 4. The library system presents a list of all matching books 5. The user selects a book. 6. The system displays the detail view for this book. 7. The user selects borrow from the menu for this book. 8. The user is already logged in. The system issues a message to the archive that the book is reserved for the user.
Extensions	4a) There are no matches to the query. 4a1) The system returns to the main screen. 8a)The user is not logged in. 8a1)The user logs in as described in Log in.
Variations	3a) The user enters the name of the author. 3b)The user selects the author from an author list. 3b1)The user clicks on "select author". 3b2)The system displays a selection list of all authors. 3b3)The user selects an author from the list.
Included Use Cases	Log in

Name	Log in
Goal	This use case describes how a library user logs into the system to prove his identity.
Preconditions	None
Postconditions	The user is logged in

Name	Borrow Book
Main Success Scenario	1. The user selects log in from the main menu. 2. The system asks the user for his login name. 3. The user enters his login name. 4. The system asks the user for his password. 5. The user enters his password. 6. The system verifies login and password. They are ok. 7. The system logs the user on.
Extensions	6a) The combination of login and password is not ok. 6a1) If the number of retries is not exceeded, repeat the use case from step 2.
Variations	None
Included Use Cases	None

In the following sections, we will describe how all elements of a use case document are mapped to a UML state machine.

Main Success Scenario

Each step in a use case corresponds to a message sent by an actor to the system or vice versa. The interval between two messages sent to the system is an abstract state of the sys-tem. As proposed by Rumbaugh [18], we denote all messages sent by the system as actions of the state. Each message sent by an actor is denoted as an event, causing a transition between two states of the system. The beginning of the use case is modelled by an initial state; the use case ends in a final state of the state machine. As intended in use case analysis, the main success scenario ends with the successful achievement of the goal [3]. Thus, the final state reached after the last step of the use case corresponds to successful completion. The whole state diagram is encapsulated in a super state named after the use case for later reuse - to model relationships to other use cases. **Fig. 1** shows the state diagram corresponding to the *Borrow Book* use case.

Each UML state diagram corresponds to an or an abstract system object, e.g. the Library. The events shown in the state diagram, e.g. *Enter Title* and *Select Search*, are not necessarily methods of a concrete class but events the system understands. These map to statements in a test script for automatic system testing.

Variations

Variations in use cases are usually local alternatives for executing a step in the use case. In a state diagram these can be modelled as multiple paths connecting two states. In the simplest case, a variant can be represented as an additional link between the two states delimiting the corresponding step in the main success scenario. Intermediate states may be needed to model more complex variants. The state chart modifications needed in our example for variants 3a) and 3b) are shown in Fig. 2.

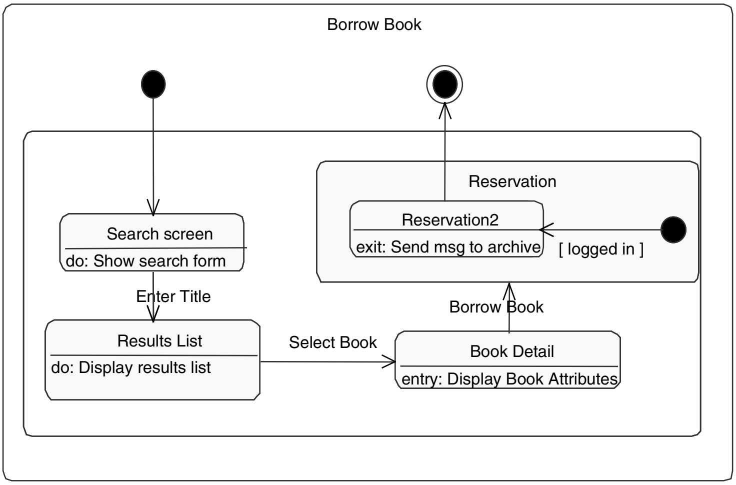


Fig. 1. Basic state diagram capturing the main success scenario.

Extensions

An extension usually describes a backup solution for completing a subgoal in a use case. A special case of this is when a subgoal (step) fails, because a precondition does not hold as in the example's extension 8a). A modular way to handle extensions is to specify them using substates of the state representing the corresponding step in the main success scenario. Extension 4a) is an exception, where we abandon the goal of the use case. No book is found and the library system goes into an error state.

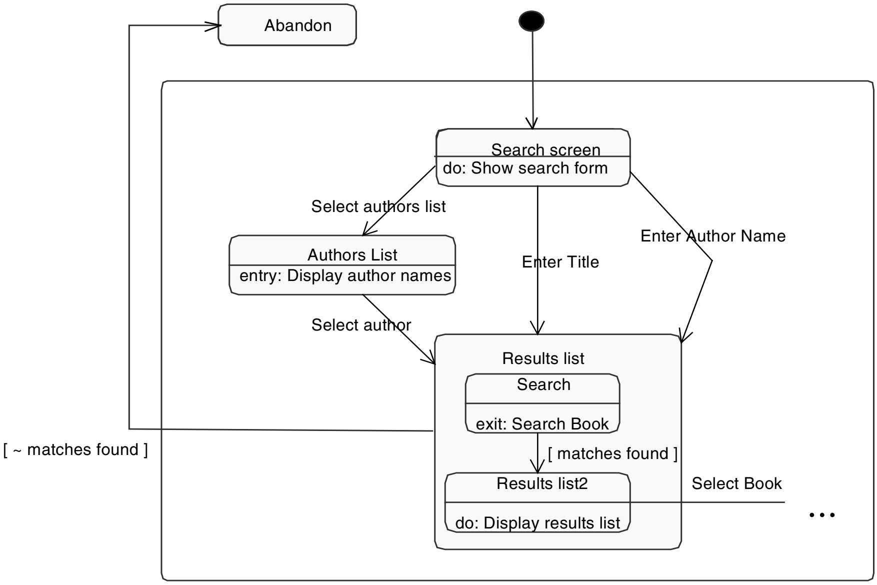


Fig. 2. State diagram with variations and extensions. The Log in state is currently a placeholder.

Preconditions and Postconditions

The Preconditions and Postconditions sections of the use case template allow us to specify the contract of the use case [14]. Preconditions describe verifiable conditions, which must hold before the execution of the use case. We model the preconditions of the use case as constraints on the first state representing the use case. The upper part of **Fig. 3** shows how a precondition on a state can be modelled in UML using a superstate with two substates.

We model the postconditions of a use case as constraints on the last state representing the use case. The lower part of **Fig. 3** shows how a postcondition on a state can be modelled using a superstate with two substates. In contrast to our previous work [7], we now model postconditions of the use case by actions which assure the postcondition of the use case (add *C*). These actions formalise the idea that the use case establishes the postconditions on successful completion. We exploit these action statements during test suite planning, when we match the actions establishing a condition with the preconditions of other use cases or steps requiring this condition.

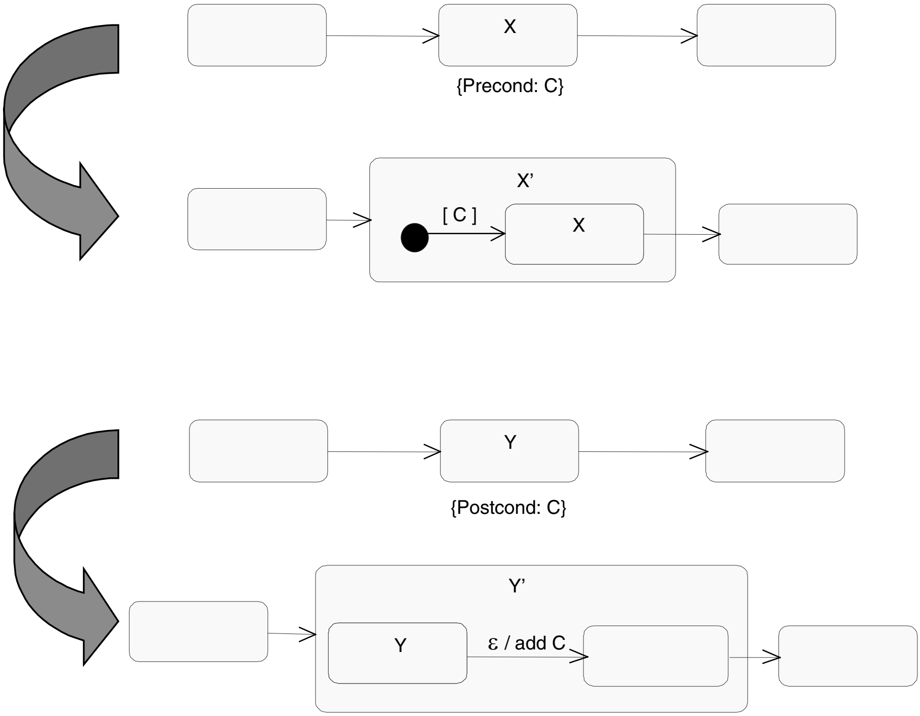


Fig. 3. Modelling Pre- and Postconditions using UML State Diagrams

Subordinate Use Cases

UML defines a mechanism for including one use case as a subfunction in another use case. This information is covered by the subordinate use case section of our template. In our example, the *Borrow Book* use case includes the functionality of the *Log In* use case. To integrate the *Log In* use case with the *Borrow Book* use case we use two techniques from the UML state diagram notation:

- A submachine reference state allows us to copy the state machine formalising the subordinate use case (*Log in*) into the enclosing use case (*Borrow Book*).
- Stub states allow us to connect the states in the submachine (*Log in*) to the right states in the enclosing machine (*Borrow Book*).

Fig. 4 shows how the use case *Log In* is embedded into *Borrow Book*.

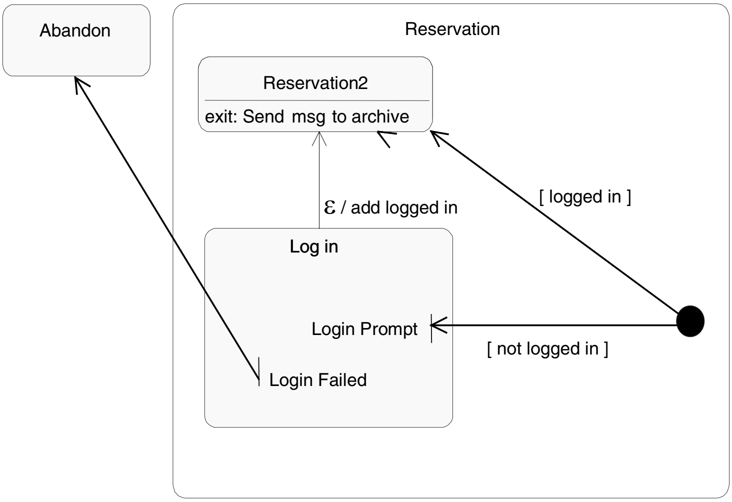


Fig. 4. Connection of Borrow Book and Log in use case.

4 From State Machines to Test Cases

Using state models to derive test cases has been common practice in the software testing world for some time [13]. The final goal of model-based testing is to automate the test case generation from test models as much as possible.

The approach presented in the current chapter takes automation further than previous approaches by interpreting the preconditions and actions of the transitions. Our algorithm generates a set of valid test sequences, where the preconditions of all transitions are established either by previous actions or by properties of the test data.

This is made possible by exploiting AI planning techniques, which allow us to systematically search for paths in the state machine, which satisfy all preconditions of the transitions. In particular, we describe the test generation problem as a STRIPS planning problem [5] and solve it with the graphplan tool [2].

The scope of our method is the generation of test sequences supplemented by constraints on the test data, as far as these can be derived from the information present in the state machine.

4.1 Properties of Considered State Models

One of the properties of our transformation from use cases to state machines is that the resulting state models have conditional transitions. As shown in the example in section 3.1, we introduce an identifier (e.g. *logged in*) for each pre- and postcondition in a use case or use case step. From now on, we will call these conditions *Propositions* and denote the set of all propositions in the state machine by A.

The propositions usually encode properties of the session or user on a semantical level. Typical propositions are *connected*, *logged in*, *items paid*, *request pending*, or *unsaved changes*. Note that the state space of a piece of software modelled in our approach consists of multiple dimensions: the state of the execution modelled by states and transitions of the state machine, and the state of the session or user modelled by propositions.

4.2 Structure of a STRIPS Planning Problem

Before we establish the connection between test generation and STRIPS planning, we have to introduce some STRIPS [5] terminology. In the search space of a STRIPS planning problem each state is described by a set of propositions which hold in that state. A set of operators describes the transitions among the states. The planning task is to find a sequence of operators which safely connects the initial state (called Σ) to the final (goal) state (called Ω). The fundamental advantage of the STRIPS language lies in the way the operators are defined.

An operator $\alpha = (Pre, Add, Del)$ is defined by

- A set *Pre* of propositions, called *Prerequisites of* α. These prerequisites must be true in every state to which α is applied.
- A set *Add* of propositions, which are true in every state resulting from the application of α, i.e. *Add* describes the propositions which are established by applying α.
- A set *Del* of propositions, which may no longer hold in a state directly resulting from the application of α. This means, every other proposition $p \in A \setminus Del$ remains true in the state resulting from application of α, if it was true before.

The power of this operator definition is that it only describes which propositions are established or possibly deleted by the application of α. All propositions not mentioned maintain their current values. This definition without the explicit description of what remains constant avoids the so-called *frame problem* [17], from which many other representations suffer. The STRIPS formalism is a widely accepted language for specifying planning problems and supported by many state of the art planning tools.

In examples of this paper, we use only propositional logic in the definitions of the operators, i.e. sets of atoms, which contain no variables. The planning tool we use allows in addition the use of free variables. With this simple extension we can define an operator *show*(X), which has *invisible*(X) as prerequisite and *visible*(X) in the Add-list. Since the planning tool requires the domain of X to be finite and defined, this is a safe use of variables, which does not confront us with the problems of first-order logic, since all variables can be instantiated based on the given domain [16].

4.3 Basic Generation of a Planning Problem

Let us first define a planning problem, which yields a test case including a selected state or a selected transition. In section 4.4 we will generalise our results leading to an algorithm for generating a set of test cases covering all states or all transitions. Consider the example shown in **Fig. 5**, which is a simplified version of the example from section 3.1. For easier reference, we have labelled each transition with a name (*t0* through *t9*).

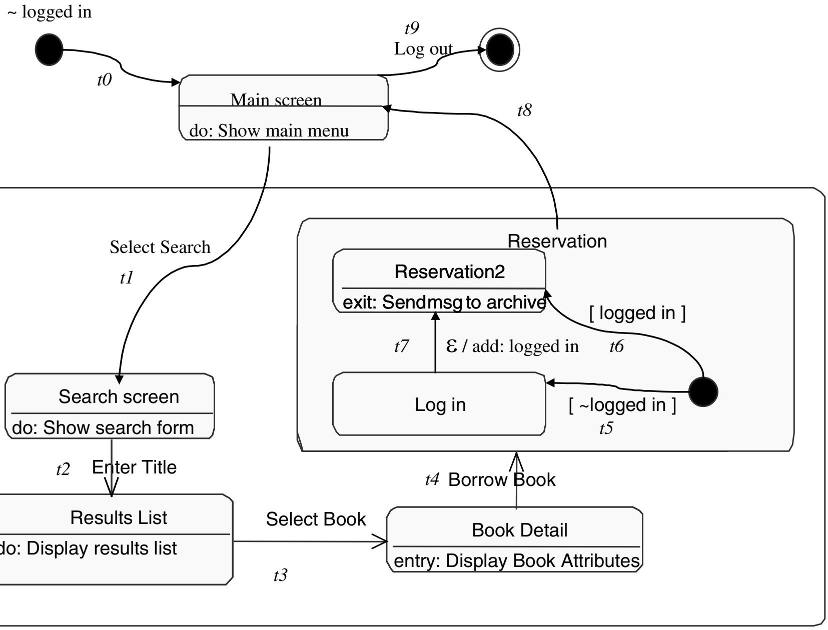


Fig. 5. Simple state machine example for test case generation.

Operator Definition

In the planning problem we represent each transition of the state machine by an operator. To specify the operators we need some additional propositions:

- For each state *s* in the state machine, we introduce two propositions:
  - a proposition *in(s)*, denoting that the current state of execution is *s* and
  - a proposition *log_state(s)*, denoting that the state *s* was the current state at some point in time.
- For each transition *t* in the state machine, we introduce a proposition *log_trans(t)*, denoting that the transition *t* was used at some time.

Now consider the transition *t* shown in **Fig. 6**, which contains all elements of a transition (event, condition, and action). We represent this transition by the operator

$$\alpha := (Pre, Add, Del),$$

where

$$
\begin{aligned}
\mathit{Pre} &= \{in(s_0), p\}\\
\mathit{Add} &= \{in(\mathrm{s}_1), log\_state(s_0), log\_trans(t), p1\}\\
\mathit{Del} &= \{in(\mathrm{s}_0)\}
\end{aligned}
$$

This operator definition models both the transitions among the states of the state machine and the value changes of propositions. The prerequisites specify that the operator can only be applied, if the current state is s_0 ($in(s_0)$) and the proposition p is true. The *Add*-list shows that after the application of the operator the state is s_1 ($in(\mathrm{s}_1)$). We create log-entries for the facts that we were in state s_0 ($log\_state(s_0)$) and that we used the transition t (log_trans(t)).

The change in the set of propositions is expressed by adding *p1*, because the proposition *p1* is made true by the action *Add: p1*. The *Del*-list shows that the current state is no longer s_0 ($in(s_0)$). The truth value of p and all other propositions not mentioned on the *Add* or *Del*-list is guaranteed to be preserved.

The operator definition is completely analogous for a transition with a *Del*-action. If the action would be *Del: p1* instead of *Add: p1* in **Fig. 6**, we would put *p1*on the *Del*-list instead of the *Add*-list. In case the condition and / or action are missing on a transition, the operator definition is the same, just without the corresponding entries in the *Pre*, *Add*, and *Del*-lists.

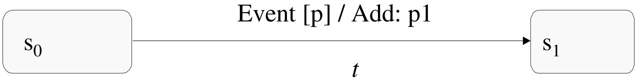


Fig. 6. Transition *t* with Event, Condition, and Action

The following table gives the operator definitions for the state machine from **Fig. 5** As a convention, we call the initial state of the whole state machine *initial*, and the final state of the whole state machine *final*. Similarly, we denote the initial state within a state *s* by *initial(s)* and the final state within a state *s* by *final(s)*. Further, we use the character ¬ to denote the negation of a proposition, e.g. ¬ *logged in* reads "*not logged in*".

Operator	Pre	Add	Del

α0	in(initial)	log_state(initial),
		log_trans(t0)
		in(Main Screen)
			in(initial)

α1	in(Main Screen)	log_state(Main Screen)
		log_trans(t1)
		in(Search Screen)
			in(Main Screen)

α2	in(Search Screen)	log_state(Search Screen)
		log_trans(t2)
		in(Results List)
			in(Search Screen)

α3	in(Results List)	log_state(Results List)
		log_trans(t3)
		in(Book Detail)
			in(Result List)

α4	in(Book Detail)	log_state(Book Detail)
		log_trans(t4)
		in(initial(Reservation))
			in(Book Detail)

Operator	Pre	Add	Del

$\alpha 5$	in(initial(Reservation))
	$\neg$ logged in
		log_state(initial(Reservation))
		log_trans(t5)
		in(Log in)
			in(initial(Reservation))

$\alpha 6$	in(initial(Reservation))
	logged in
		log_state(initial(Reservation))
		log_trans(t6)
		in(Reservation2)
			in(initial(Reservation))

$\alpha 8$	in(Reservation2)	log_state(Reservation2)
		log_trans(t8)
		in(Main Screen)
			in(Reservation2)

$\alpha 9$	in(Main Screen)	log_state(Main Screen)
		log_trans(t9)
		in(final)
			in(Main Screen)
			logged in

Initial State

Having the operator definitions in place, we have to specify the initial conditions and the goal in order to arrive at a complete planning problem, which can be solved by a planning tool. The initial state Σ of our planning problem specifies that we start the execution of a test in the initial state of the state machine. Moreover, the initial state has to specify for each proposition $p \in A$, if it is true of false in the beginning of the execution. For propositions, which keep track of the program execution, we will always assume that they are false in the initial state. Thus, in our small example, we assume that the proposition *logged in* is false. Based on these considerations, we have

$$\Sigma = \{\neg \textit{logged in}, \textit{in}(\textit{initial})\}.$$

Goal

We have included *log-state* and *log-trans*-entries in the operator definitions to keep track of the states and transitions covered during the execution of the test case. By using these entries in the formulation of the goal state, we can instruct the planner to create test cases, which include certain states and transitions. As an example, let us derive a test case, which tests the transition *t6*, which leads to the *Reservation2* state, in case the user is already logged in. The corresponding goal statement is

$$\Omega = \{\textit{log-trans}(\textit{t6}), \textit{in}(\textit{final})\}.$$

The presence of the proposition *in*(*final*) makes sure, that the test case ends in the final state of the state machine.

Computation of the Test Case

In Appendix A, we show the graphplan input files corresponding to the planning problem described in the previous sections. The graphplan execution trace shown at the end of Appendix A points out that graphplan generates the desired test sequence in 0.06s. The test sequence consists of 15 steps (borrow two books in sequence, so that the second time the user is already logged in).

This small proof of concept example had the goal to illustrate the basic steps in test sequence generation using a planning tool. In the following section, we will derive a systematic algorithm for generating a set of test sequences with a given coverage level.

4.4 Algorithms for Achieving a given Coverage Level

Properties of the Test Data

Let us now reconsider the meaning of our propositions: Propositions represent the pre- and postconditions of the use cases. In our previous example, the proposition *logged in* represented the fact that the user has already logged into the system. This condition can be achieved through successful completion of the Log *in* use case. In general however, we cannot assume that all conditions are achievable by use cases. They can as well represent properties of the test data / test input. As an example, consider **Fig. 7**: In this modified version of our state machine example, we have added a check whether the selected book is available and expressed it by the proposition *Book available*. Obviously, there exists no transition in the state model which makes *Book available* true. The truth value of *Book available* depends entirely on the title of the book, which the user enters. Thus, we cannot guarantee that the book will be available only by the design of a test sequence, but we have to record that the test input must have this property. We assure this using a two-step approach:

1. We determine all propositions in the state machine which do not occur in an *add* or *del*-action.
2. For each proposition p found in step 1, we create two operators:

$$\alpha_p^+ = (Pre, Add, Del), \text{ where } Pre = \varnothing, Add = \{p\}, Del = \varnothing$$
$$\alpha_p^- = (Pre, Add, Del), \text{ where } Pre = \varnothing, Add = \{\neg p\}, Del = \varnothing$$

In the example from **Fig. 7**, this procedure adds an operator which makes *Book available* true and one, which makes it false. Before e.g. transition *t8* can be passed, the operator, which makes *Book available* true must be applied. From the plan, our algorithm can see that this pseudo-operator has been applied and derive that the test input has to have the property *Book available*. Thus, in addition to the test sequence, we also formally deduce a set of constraints for the test data (in this case the availability of the book).

Test Coverage

The weakest coverage criterion for state machine testing is to cover every state. The next better coverage level, which is equivalent to branch coverage, is to cover every transition. This is proposed e.g. in [13]. To achieve this coverage level, we construct the test suite iteratively. We start by marking every transition as *not covered*. Then, we generate the first test sequence and mark every transition in the test sequence as *covered*. Then, in a loop, we pick one transition not yet covered and create a test sequence, which includes this transition (and maybe some further previously not covered transitions). We have described how to create such a test sequence including one selected transition in section 4.3. We repeat this procedure until all transitions are covered by at least one test sequence.

A complication of the above mentioned procedure arises in the case of nested state machines. For example, in our state machines from **Fig. 5** and **Fig. 7**, *Log in* is a sub-machine reference state, representing the state machine resulting from the *Log in* use case. There are two ways to handle test coverage for sub-machine reference states.

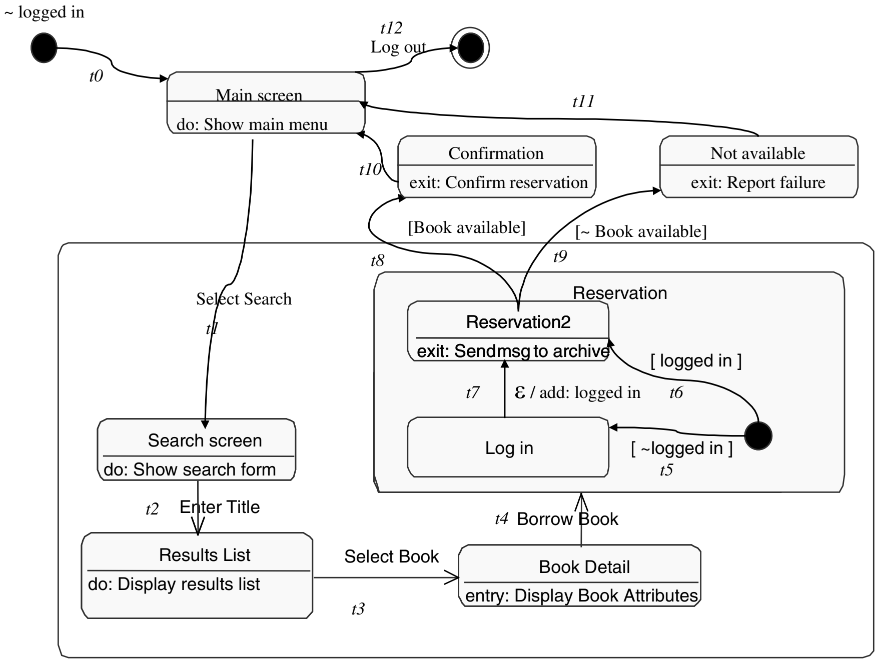


Fig. 7. Modified state machine example. It adds the possibility that the book selected by the library user is not available (e.g. already borrowed by someone else).

- Test separately: If the sub-machine can be tested separately, we can generate a test suite for the sub-machine and ignore coverage for the sub-machine, when designing a test suite for the surrounding state machine. In our example, this may be a sensible choice, because the *Log in* procedure does not depend on the details of the book being borrowed and can thus probably be tested separately.
- Combine test sequences: In the general case, we cannot assume, that the sub-machine is tested separately. For example, if we had a sub use case describing the communication of the library computer with the central database server it would probably be beneficial to test this together with the surrounding borrow book use case, because the communication depends on the book data provided by the user.

We achieve test coverage for the surrounding and included state machines by the following recursive procedure:

- Generate a test suite for the included state machine and record the number *n* of test cases in this suite.
- Generate a test suite for the surrounding state machine with the additional constraint that the sub-machine reference state must occur at least *n* times in the test cases.
- "Folding in": Insert the *n* different test sequences for the sub-machine at the occurrences of the sub-machine reference state in the test sequences for the surrounding state machine.

Algorithm

The considerations of the previous two sections lead to the following algorithm for test suite generation:

Algorithm	Create Test Suite
Input	A state machine *M*.
Output	A list of test sequences and constraints on the test data for each test sequence.
Preprocessing	• Generate the set of operator definitions for *M*. • Collect the set *A* of all propositions occurring in the state machine. • For each proposition $p \in A$: Check if *p* occurs in an action statement of the form *Add*:*p* or *Del*:*p*. If not, generate two new operators for *p*, for recording it as a constraint on the test data: $\alpha_p^+ = (\emptyset, \{test\text{-}data(p), p\}, \emptyset\}$, $\alpha_p^- = (\emptyset, \{test\text{-}data(\neg p), \neg p\}, \emptyset\}$ Create the initial state definition: $\Sigma = \cup_{p \in A} \{\neg p\} \cup \{in(initial)\}$ • Create a list of all transitions and mark each transition as not covered.
Sub-machines	IF sub-machines exist: 1. Create a counter for each state indicating, how often this state must occur in the test suite. 2. Initialise all counters to 1. 3. For each (first occurrence of a) sub-machine reference state: • Call *Create Test Suite* for the sub-machine and record the results. • Set the minimum occurrence counter of the sub-machine reference state to the number of test sequences in the test suite.
Test Generation	WHILE unmarked transitions exist OR an occurrence counter > 0 for any state exists: 1. Select (randomly) an unmarked transition or a state with occurrence counter > 0. 2. Generate the goal definition for a test sequence covering the selected transition or state. 3. Call the planner to obtain the test sequence. 4. Mark all transitions covered by the plan. 5. Decrease the occurrence counters of all states (if they exist) by the number of occurrences of the state in the test sequence. 6. Derive the constraints of the test data from the pseudo operators in the plan. Combine the obtained test sequences with the sequences form the sub-machines. Return the test suite consisting of the test sequences together with the constraints on the test data.

5 Conclusions

In our paper we have discussed a new approach to automatically generate test cases from use cases. We do that in two steps:

1. Formal transformation of a detailed use case description including pre- and postconditions to a UML state model
2. Generation of test cases from the state model

Our formal transformation from use cases was introduced in [7] and was presented in a shortened and slightly revised version here. Our test case generation approach transforms the problem at hand into a planning problem and uses STRIPS - the most widely used AI planning formalism – to derive a test suite for a given state chart. Using this planning technique ensures that the test sequences derived from the state machine are consistent in the sense that the preconditions of all transitions in the sequence are satisfied. This is an improvement over previous methods which generate arbitrary paths and do not consider conditional transitions. First results using the state-of-the-art planner graphplan show the feasibility of our approach. Our method ensures, that a test suite is generated for the system test, which is consistent with requirements and the results of requirements analysis (the UML state model).

With our approach we now have a coherent procedure to derive test cases from use cases in a formal and partly automatic way. Our procedure can however not prevent that some manual additions have to be made:

- The expected system responses have to be added to the test sequence manually to yield complete test cases.
- Although we derive some constraints on the test inputs automatically, the concrete test data still has to be defined manually.

Ongoing and further work concentrates on extensions to our approach: Stronger coverage criteria will be considered. Besides, systematic consideration should be given to other aspects of the specification, e.g. performance and frequency requirements.

References

[1] Robert Binder. *Testing Object-Oriented Systems*. Addison-Wesley, 1999.

[2] A. Blum, M. Furst. *Fast Planning Through Planning Graph Analysis*. Artificial Intelligence, 90:281-300 (1997).

[3] Alistair Cockburn. *Structuring Use Cases with Goals.* Journal of Object-Oriented Programming, Sep/Oct, 1997, pp. 35-40, and Nov/Dec, 1997, pp. 56-62.

[4] Jules Desharnais, Marc Frappier, Ridha Khèdri, and Ali Mili. *Integration of Sequential Scenarios.* IEEE Transaction on Software Engineering, Vol. 24, No. 9, September 1998.

[5] R.E. Fikes, N.J. Nilsson. *STRIPS: a new approach to the application of theorem proving to problem solving.* Artificial Intelligence 2, 1971.

[6] Martin Fowler. *Use and Abuse Cases.* Distributed Computing, April 1998.

[7] Peter Fröhlich, Johannes Link: *Modelling Dynamic Behaviour Based on Use Cases*. Proceedings of Quality Week Europe, Brussels, November 1999.

[8] D. Harel. *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, Vol. 8, 1987.

[9] D. Harel. *On Visual Formalisms*. Communications of the ACM, Vol. 31, No. 5, Pages 514-531, May 1988.

[10] D. Harel, Michael Politi. *Modeling Reactive Systems With Statecharts: The STATEMATE Approach*. McGraw-Hill, New York, N.Y., 1998.

[11] Ivar Jacobson, Magnus Christerson, Patrick Jonsson, Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Wokingham, England, 1992.

[12] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[13] Brian Marick. *The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing*. Prentice Hall, 1995.

[14] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[15] Object Management Group. *Unified Modeling Language Specification*. Framingham, Mass., 1998.

[16] R. Reiter. Equality and Domain Closure in First Order Databases. Journal of the ACM, 32:57–97, 1987.

[17] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In: V. Lifshitz (Ed.), *Artificial Intelligence and mathematical theory of computation: Papers in honor of John McCarthy*. Boston: Academic Press, 1991.

[18] James Rumbaugh, Michel Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, N.J., 1991.

[19] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999.

[20] Russell R. Hurlbut. *The Three R's of Use Case Formalisms: Realization, Refinement, and Reification.*Technical Report XPT-TR-97-06, Expertech, Ltd, 1997.

[21] Geri Schneider, Jason P. Winters. *Applying Use Cases: A Practical Guide.* Addison-Wesley, 1998.

Appendix A: Graphplan Operator Definition File

```
(operator alpha0
        (params)
        (preconds (in initial))
        (effects (del in initial)
                 (log-state initial) (log-trans t0)
                 (in main-screen)))

(operator alpha1
        (params)
        (preconds (in main-screen))
        (effects (del in main-screen)
                (log-state main-screen)
                (log-trans t1)
                (in search-screen)))

(operator alpha2
        (params)
        (preconds (in search-screen))
        (effects (del in search-screen)
                (log-state search-screen) (log-trans t2)
                (in results-list)))

(operator alpha3
        (params)
        (preconds (in results-list))
        (effects (del in results-list)
                (log-state results-list) (log-trans t3)
                (in book-detail)))

(operator alpha4
        (params)
        (preconds (in book-detail))
        (effects (del in book-detail)
                (log-state book-detail) (log-trans t4)
                (in initial-reservation)))
(operator alpha5
        (params)
        (preconds (in initial-reservation) (not-logged-in))
        (effects (del in initial-reservation)
                (log-state initial-reservation) (log-trans t5)
                (in log-in)))

(operator alpha6
        (params)
        (preconds (in initial-reservation) (logged-in))
        (effects (del in initial-reservation)
                (log-state initial-reservation) (log-trans t6)
                (in reservation2)))

(operator alpha7
        (params)
        (precond (in log-in))
        (effects (del in log-in) (del not-logged-in)
                 (log-state log-in) (log-trans t7)
                 (in reservation2)
                 (logged-in)))

(operator alpha8
        (params)
        (precond (in reservation2))
        (effects (del in reservation2)
                 (log-state reservation2) (log-trans t8)
                 (in main-screen)))

(operator alpha9
        (params)
        (precond (in main-screen))
        (effects (del in main-screen) (del logged-in)
                 (log-state main-screen) (log-trans t9)
                 (in final) (not-logged-in)))
```

Initial State and Goal:

```
(preconds
       (not-logged-in)
       (in initial))

(effects
       (log-trans t6)
       (in final))
```

Execution trace of graphplan:

```
alpha0
alpha1
alpha2
alpha3
alpha4
alpha5
alpha6
alpha7
alpha8
alpha9
facts loaded.
time: 1, 5 facts and 3 exclusive pairs.
time: 2, 10 facts and 17 exclusive pairs.
time: 3, 13 facts and 32 exclusive pairs.
time: 4, 16 facts and 50 exclusive pairs.
time: 5, 19 facts and 71 exclusive pairs.
time: 6, 22 facts and 95 exclusive pairs.
time: 7, 26 facts and 135 exclusive pairs.
time: 8, 28 facts and 142 exclusive pairs.
time: 9, 28 facts and 99 exclusive pairs.
time: 10, 28 facts and 88 exclusive pairs.
```

```
time: 11, 28 facts and 79 exclusive pairs.
time: 12, 28 facts and 72 exclusive pairs.
time: 13, 29 facts and 76 exclusive pairs.
time: 14, 29 facts and 75 exclusive pairs.
time: 15, 29 facts and 71 exclusive pairs.
Goals first reachable in 15 steps.
765 nodes created.
goals at time 16:
  log-trans_t6 in_final
1 alpha0
2 alpha1
3 alpha2
4 alpha3
5 alpha4
6 alpha5
7 alpha7
8 alpha8
9 alpha1
10 alpha2
11 alpha3
12 alpha4
13 alpha6
14 alpha8
15 alpha9
0 entries in hash table,
14 total set-creation steps (entries + hits + plan length - 1).
15 actions tried
  0.06 secs
```

Author Index